

21世纪全国应用型本科计算机案例型规划教材




数据结构

（C语言版）

主 编 陈超祥



- 强调理论与实践相结合，注重培养专业技能
- 采用“问题导入”和“任务驱动”编写方式
- 引入企业工程师参与，理论与实践紧密结合

北京大学出版社
PEKING UNIVERSITY PRESS

21 世纪全国应用型本科计算机案例型规划教材

数据结构(C 语言版)

主　编　陈超祥
副主编　徐　萍　李文书
参　编　刘静静　王卫东　曹　冶
　　　　黄科海　周　苗

北京大学出版社
PEKING UNIVERSITY PRESS

内 容 简 介

本书是为“数据结构”课程编写的教材，共9章，系统介绍了各种常用的数据结构与算法方面的基本知识。第1章为绪论，引入了数据结构与算法的一些基本概念；第2～7章分别介绍了线性表、栈、队列、串、多维数组、树和图等几种基本的数据结构；第8章和第9章分别介绍了多种查找和排序的算法。

本书引入的主要案例都源自实际项目应用，案例、项目由企业工程师根据章节内容设计并实现，全部程序都在C Free 5.0中调试通过。

本书可以作为高等院校计算机、软件工程等相关专业本科学生的教材，也可以作为其他理工科专业的选修教材，还可供从事计算机应用的工程技术人员参考，读者只需掌握C语言编程的基本技术就可以学习本书。

图书在版编目(CIP)数据

数据结构. C语言版/陈超祥主编. —北京：北京大学出版社，2013.8

(21世纪全国应用型本科计算机案例型规划教材)

ISBN 978-7-301-22965-1

Ⅰ. ①数… Ⅱ. ①陈… Ⅲ. ①数据结构—高等学校—教材②C语言—程序设计—高等学校—教材 Ⅳ. ①TP311.12②TP312

中国版本图书馆CIP数据核字(2013)第179599号

书　　名：数据结构(C语言版)
著作责任者：陈超祥　主编
策 划 编 辑：郑　双
责 任 编 辑：郑　双
标 准 书 号：ISBN 978-7-301-22965-1/TP · 1300
出 版 发 行：北京大学出版社
地　　址：北京市海淀区成府路205号　100871
网　　址：http://www.pup.cn　　新浪官方微博：@北京大学出版社
电 子 信 箱：pup_6@163.com
电　　话：邮购部62752015　发行部62750672　编辑部62750667　出版部62754962
印 刷 者：
发 行 者：北京大学出版社
经 销 者：新华书店
787毫米×1092毫米　16开本　16.75印张　380千字
2013年8月第1版　2018年7月第3次印刷
定　　价：32.00元

21世纪全国应用型本科计算机案例型规划教材

专家编审委员会

信息技术的案例型教材建设

(代丛书序)

刘瑞挺

北京大学出版社第六事业部在 2005 年组织编写了《21 世纪全国应用型本科计算机系列实用规划教材》，至今已出版了 50 多种。这些教材出版后，在全国高校引起热烈反响，可谓初战告捷。这使北京大学出版社的计算机教材市场规模迅速扩大，编辑队伍茁壮成长，经济效益明显增强，与各类高校师生的关系更加密切。

2008 年 1 月北京大学出版社第六事业部在北京召开了“21 世纪全国应用型本科计算机案例型教材建设和教学研讨会”。这次会议为编写案例型教材做了深入的探讨和具体的部署，制定了详细的编写目的、丛书特色、内容要求和风格规范。在内容上强调面向应用、能力驱动、精选案例、严把质量；在风格上力求文字精练、脉络清晰、图表明快、版式新颖。这次会议吹响了提高教材质量第二战役的进军号。

案例型教材真能提高教学的质量吗？

是的。著名法国哲学家、数学家勒内·笛卡儿(Rene Descartes，1596—1650)说得好：“由一个例子的考察，我们可以抽出一条规律。(From the consideration of an example we can form a rule.)”事实上，他发明的直角坐标系，正是通过生活实例而得到的灵感。据说是在 1619 年夏天，笛卡儿因病住进医院。中午他躺在病床上，苦苦思索一个数学问题时，忽然看到天花板上有一只苍蝇飞来飞去。当时天花板是用木条做成正方形的格子。笛卡儿发现，要说出这只苍蝇在天花板上的位置，只需说出苍蝇在天花板上的第几行和第几列。当苍蝇落在第四行、第五列的那个正方形时，可以用(4，5)来表示这个位置……由此他联想到可用类似的办法来描述一个点在平面上的位置。他高兴地跳下床，喊着“我找到了，找到了”，然而不小心把国际象棋撒了一地。当他的目光落到棋盘上时，又兴奋地一拍大腿：“对，对，就是这个图”。笛卡儿锲而不舍的毅力，苦思冥想的钻研，使他开创了解析几何的新纪元。千百年来，代数与几何，井水不犯河水。17 世纪后，数学突飞猛进的发展，在很大程度上归功于笛卡儿坐标系和解析几何学的创立。

这个故事，听起来与阿基米德在浴缸洗澡而发现浮力原理，牛顿在苹果树下遇到苹果落到头上而发现万有引力定律，确有异曲同工之妙。这就证明，一个好的例子往往能激发灵感，由特殊到一般，联想出普遍的规律，即所谓的“一叶知秋”、“见微知著”的意思。

回顾计算机发明的历史，每一台机器、每一颗芯片、每一种操作系统、每一类编程语言、每一个算法、每一套软件、每一款外部设备，无不像闪光的珍珠串在一起。每个案例都闪烁着智慧的火花，是创新思想不竭的源泉。在计算机科学技术领域，这样的案例就像大海岸边的贝壳，俯拾皆是。

事实上，案例研究(Case Study)是现代科学广泛使用的一种方法。Case 包含的意义很广：包括 Example 例子，Instance 事例、示例，Actual State 实际状况，Circumstance 情况、事件、境遇，甚至 Project 项目、工程等。

我们知道在计算机的科学术语中，很多是直接来自日常生活的。例如 Computer 一词早在 1646 年就出现于古代英文字典中，但当时它的意义不是“计算机”而是“计算工人”，即专门从事简单计算的工人。同理，Printer 当时也是“印刷工人”而不是“打印机”。正是

由于这些“计算工人”和“印刷工人”常出现计算错误和印刷错误，才激发查尔斯·巴贝奇(Charles Babbage，1791—1871)设计了差分机和分析机，这是最早的专用计算机和通用计算机。这位英国剑桥大学数学教授、机械设计专家、经济学家和哲学家是国际公认的“计算机之父”。

20 世纪 40 年代，人们还用 Calculator 表示计算机器。到电子计算机出现后，才用 Computer 表示计算机。此外，硬件(Hardware)和软件(Software)来自销售人员。总线(Bus)就是公共汽车或大巴，故障和排除故障源自格瑞斯·霍普(Grace Hopper，1906—1992)发现的“飞蛾子”(Bug)和“抓蛾子”或“抓虫子”(Debug)。其他如鼠标、菜单……不胜枚举。至于哲学家进餐问题，理发师睡觉问题更是操作系统文化中脍炙人口的经典。

以计算机为核心的信息技术，从一开始就与应用紧密结合。例如，ENIAC 用于弹道曲线的计算，ARPANET 用于资源共享以及核战争时的可靠通信。即使是非常抽象的图灵机模型，也受益于二战时图灵博士破译纳粹密码工作的关系。

在信息技术中，既有许多成功的案例，也有不少失败的案例；既有先成功而后失败的案例，也有先失败而后成功的案例。好好研究它们的成功经验和失败教训，对于编写案例型教材有重要的意义。

我国正在实现中华民族的伟大复兴，教育是民族振兴的基石。改革开放 30 年来，我国高等教育在数量上、规模上已有相当的发展。当前的重要任务是提高培养人才的质量，必须从学科知识的灌输转变为素质与能力的培养。应当指出，大学课堂在高新技术的武装下，利用 PPT 进行的“高速灌输”、“翻页宣科”有愈演愈烈的趋势，我们不能容忍用“技术”绑架教学，而是让教学工作乘信息技术的东风自由地飞翔。

本系列教材的编写，以学生就业所需的专业知识和操作技能为着眼点，在适度的基础知识与理论体系覆盖下，突出应用型、技能型教学的实用性和可操作性，强化案例教学。本套教材将会有机融入大量最新的示例、实例以及操作性较强的案例，力求提高教材的趣味性和实用性，打破传统教材自身知识框架的封闭性，强化实际操作的训练，使本系列教材做到“教师易教，学生乐学，技能实用”。有了广阔的应用背景，再造计算机案例型教材就有了基础。

我相信北京大学出版社在全国各地高校教师的积极支持下，精心设计，严格把关，一定能够建设出一批符合计算机应用型人才培养模式的、以案例型为创新点和兴奋点的精品教材，并且通过一体化设计、实现多种媒体有机结合的立体化教材，为各门计算机课程配齐电子教案、学习指导、习题解答、课程设计等辅导资料。让我们用锲而不舍的毅力，勤奋好学的钻研，向着共同的目标努力吧！

刘瑞挺教授 本系列教材编写指导委员会主任、全国高等院校计算机基础教育研究会副会长、中国计算机学会普及工作委员会顾问、教育部考试中心全国计算机应用技术证书考试委员会副主任、全国计算机等级考试顾问。曾任教育部理科计算机科学教学指导委员会委员、中国计算机学会教育培训委员会副主任。PC Magazine《个人电脑》总编辑、CHIP《新电脑》总顾问、清华大学《计算机教育》总策划。

前　　言

数据结构是计算机科学中最重要的课程之一，是计算机科学的算法理论基础和程序设计的技术基础，因此，数据结构是计算机及其相关专业的一门核心课程。

对于初学者，数据结构的学习有两大难题：一是抽象的数据结构原理与方法难以理解并掌握，二是难以用抽象的原理与方法来解决实际问题。一般数据结构书籍使用伪代码，不提供完整算法，给初学者带来不便。本书每章的内容通过一个实际项目来引入，结合理论知识，把项目进行完整的实现，将抽象的原理、方法与实践紧密结合，加深了读者对数据结构和算法的理解，从而提高读者的编程能力。

本书引入的主要案例都源自实际项目应用，本书编者团队由来自学校的教师和来自企业的工程师组成。理论算法和文字内容由学校教师编写，案例、项目由企业工程师根据章节内容设计并实现，全部程序都在 C Free 5.0 中调试通过。

本书共 9 章。第 1 章为绪论；第 2～4 章分别介绍了线性表、栈、队列和串等几种基本的数据结构，都属于线性结构；第 5～7 章分别介绍了多维数组、树和图等非线性结构；第 8 章和第 9 章分别介绍了查找和排序，它们都是数据处理中广泛使用的技术。

本书在确定教材体系和主要内容的过程中，得到了亚信科技杭州研发中心的张继洲、丁海涛及杭州东忠科技集团的朱丹俊等多位工程师以及浙江树人大学朱斌老师的帮助。浙江树人大学陈超祥老师编写了第 1、2 章，浙江理工大学李文书老师编写了第 3、4 章，浙江树人大学刘静静老师编写了第 5、6 章，浙江树人大学徐萍老师编写了第 7、8、9 章。本书的主要项目由浙大网新科技股份有限公司的王卫东、曹治两位工程师设计完成。全书最后由陈超祥、徐萍老师统稿。本书的编写还得到了浙大网新科技股份有限公司赵育新总监、黄科海总经理、周迪经理和杭州东忠科技集团黄炜董事长和周苗总经理的大力支持和指导，在此向他们深表感谢。

由于编者水平有限，本书难免存在一些不足之处，敬请各位专家和广大读者谅解并批评指正。

陈超祥

2013 年 4 月

目　　录



绪　论



学生信息查询问题

某学校要开发一个查询某个学生信息的程序。要求对于任意给出的一个学生学号，若该学号存在，则迅速找到其姓名、性别、专业、班级、联系方式等信息，否则指出没有该学号的学生。

1.1　为什么要学习数据结构

在计算机发展初期，人们使用计算机主要是处理数值计算问题，所涉及的运算对象是简单的整型、实型或布尔型数据，所以程序设计者的主要精力集中于程序设计的技巧上，而无需重视数据结构。随着计算机软、硬件的发展和应用领域的扩大，非数值计算问题越来越多，越来越重要，这类问题涉及的数据对象更为复杂，数据元素之间的相互关系一般无法用数学方程式加以描述。因此，在计算机中如何有效地组织和处理数据就成了迫切需要解决的问题。

1. 解决问题的方法

在编写查找学生信息的程序(本书使用 C 语言)时，一般会经历如下几个步骤。

(1) 构造一张学生信息表(即数据结构)。表中每个节点存放以下数据项：学号、姓名、性别、专业班级、联系方式，如表 1.1 所示。

表 1.1　某高校学生信息表

学号	姓名	性别	专业班级	联系方式
201105215101	许林	男	计算机应用 111 班	88222222
201105215223	李志强	男	计算机应用 112 班	13958111111
201105214110	陈敏	女	电子信息工程 111 班	13779000000
201105216125	赵小路	女	电子商务 111 班	13738555555
……	……	……	……	……

(2) 将学生信息表存储到计算机中，即确定存储结构。可以先定义一个 C 语言的结构体类型，如下所示。

```
typedef struct
{
  char number[12];
  char name[10];
  char sex;
  char class[20];
  char relation[12];
}StudentInfo;
```

再选择一维数组来储存学生的信息，该数组的存储结构如图 1.1 所示。

stu[0]	201105215101	许林	男	计算机应用111班	88222222
stu[1]	201105215223	李志强	男	计算机应用112班	13958111111
stu[2]	201105214110	陈敏	女	电子信息工程111班	13779000000
stu[3]	201105216125	赵小路	女	电子商务111班	13738555555
……	……	……	……	……	……

图 1.1　学生信息表的顺序存储结构示意图

(3) 确定算法。根据问题的要求，确定实现查找的算法。算法的实现是根据存储结构决定的，如果将学生信息表数据顺序地存储在计算机中，则本例将从头开始依次查找学生学号，直到找出与之对应的学号，然后再将其对应学生的信息显示出来。

(4) 根据算法，写出实现问题的代码。

2. 分析

在上面例子中，用数组存储了学生信息表，数组决定了计算机存储的学生信息的最大个数是固定的，这不符合实际的使用情况；数组在内存中顺序存储决定了逻辑上相邻的数据元素在物理位置上也相邻。因此，在顺序结构中查找任何一个位置上的数据元素非常方便，这是顺序存储的优点。但在对顺序结构进行插入和删除时，需要通过移动数据元素来实现，这在数据量不大的情况下是可行的，但当有成千上万的数据信息时就不实用了，将会影响程序的运行效率。



问题描述

田径赛的时间安排问题

假设某校的田径选拔赛有六个比赛项目，规定每个选手最多可参加三个项目，有五人报名参加比赛，如表 1.2 所示。设计比赛日程表，使其在尽可能短的时间内完成比赛。

表 1.2　参赛选手比赛项目表

姓名	项目 1	项目 2	项目 3
赵昕伊	跳高	跳远	100m
王东旭	标枪	铅球	

续表

姓名	项目 1	项目 2	项目 3
张涵	标枪	100m	200m
赵天怡	铅球	200m	跳高
陈瑞	跳远	200m	

1. 解决问题的方法

在解决该问题时，一般会经历如下几个步骤。

(1) 选择一个合适的数据结构来表示。

① 假设用如下六个不同的代号代表不同的项目：

跳高	跳远	标枪	铅球	100m	200m
A	B	C	D	E	F

则表 1.2 可转换为表 1.3 所示。

表 1.3 参赛选手比赛项目代号表

姓名	项目 1	项目 2	项目 3
赵昕伊	A	B	E
王东旭	C	D	
张涵	C	E	F
赵天怡	D	F	A
陈瑞	B	F	

显然，同一选手选择的几个项目间不能同一时间比赛。

② 以顶点代表比赛项目，在不能同时进行比赛的项目之间连上一条边，由此可以得到一个图，如图 1.2 所示，该图就是本问题的数据结构模型。从图中可以看出，同一个选手选择的几个项目是不能在同一时间内比赛的，因此该选手所选择的项目中应该两两有边相连。

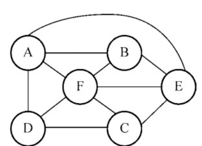


图 1.2 安排比赛项目的数据结构模型

上述由顶点和边组成的无向图是数据结构中非线性数据结构中的一类。

(2) 将该图存储到计算机中，即确定存储结构。

(3) 确定算法。竞赛项目的时间安排问题可以抽象为对该无向图进行“着色”问题，即使用尽可能少的颜色给图中每个顶点着色，使得任意两个有边连接的相邻顶点着上不同的颜色。每一种颜色表示一个比赛时间，着上同一种颜色的顶点可以安排在同一时间内比赛，如 A 和 C、B 和 D。比赛项目的时间安排如表 1.4 所示：



表 1.4　比赛项目安排表

比赛时间	比赛项目
1	A，C
2	B，D
3	E
4	F

(4) 根据算法，写出解决问题的程序。关于图的算法实现在后面的章节中将讲到，在此对本例的程序不做描述，感兴趣的读者可以自行编写。

2. 分析

上述例子是数据结构的典型例子，例子中的数据量相对较少，当在对数据比较大的问题，如校运动会、奥运会等比赛项目进行安排时，更凸显其解决问题的高效和精确。

通过对上面两个问题的分析，可以得出，为了有效地在计算机上解决具有各种数据的实际问题，首先必须研究数据及数据之间的关系(即数据结构)及对这些数据可以进行的操作(即算法)，然后还要研究具有结构关系的数据在计算机内部的存储结构以及在计算机中处理这样的存储结构的算法，以找出最适合解决问题的方案，这就是数据结构和算法所要解决的问题。该过程可表示为：

实际问题→数据结构(逻辑结构)→存储到计算机(存储结构)→编写出程序实现运算(算法)→解决问题。

1.2　数据结构概述

1. 数据

数据(Data)是指所有能输入到计算机中并被计算机程序加工、处理的符号的总称。它是计算机程序加工的“原料”，包括整数、实数、字符、声音、图形、图像等。

2. 数据元素

数据元素(Data Element)是数据的基本单位。有些情况下，数据元素也称为元素、节点、顶点、记录。有时，一个数据元素可以由若干个数据项组成，数据项是数据表示中不可分割的最小单位。图 1.3 所示为学生信息数据。其中，一个学生的信息由一个数据元素表示，一个学生信息的数据元素由学号、姓名、性别、专业班级、联系方式五个数据项组成。

学号	姓名	性别	专业班级	联系方式
201105215101	许林	男	计算机应用111班	88222222
201105215223	李志强	男	计算机应用112班	13958111111
201105214110	陈敏	女	电子信息工程111班	13779000000
201105216125	赵小路	女	电子商务111班	13738555555
……	……	……	……	……

一个数据元素

一个数据项

图 1.3　数据项和数据元素

3. 数据类型

数据类型(Data Type)是具有相同性质的数据的集合以及在这个集合上的一组操作。如整型，它是[−maxint，maxint]区间上的整数(maxint 是依赖于所使用的计算机及语言的最大整数)，在这个整数集上可以进行加、减、乘、整除、取模等操作。

数据类型可以分为原子数据类型和结构数据类型。原子数据类型由计算机语言自身提供，如 C 语言的整型、实型、字符型等；结构数据类型是借用计算机语言提供的能描述数据元素之间逻辑关系的一种机制，由用户自己根据需要定义，如 C 语言的数组、结构体等。

4. 数据结构

数据结构(Data Structure)是指数据之间的相互关系，即数据的组织形式。通常可以用一个二元组来表示。例如：

```
Data_Stucture=(D,R)
```

其中，D 是数据的有限集，R 是 D 上关系的有限集。

广义地，数据结构是指按某种逻辑关系组织起来的一批数据，应用计算机语言并按一定的存储表示方式把它们存储在计算机的存储器中，并在其上定义了一个运算的集合。数据结构一般包含以下三方面的内容。

(1) 数据元素之间的逻辑关系，也称为数据的逻辑结构。

(2) 数据元素及其关系在计算机存储器内的表示，也称为数据的存储结构。

(3) 对数据施加的操作，即数据的运算。

数据的逻辑结构是从逻辑关系上描述数据的，是数据本身所固有的，它与数据的存储无关，是独立于计算机的。数据的存储结构是逻辑结构用计算机语言的实现，是依赖于计算机的。数据的运算定义在逻辑结构上，并在存储结构上实现。每种逻辑结构都有一个运算的集合，例如，最常用的运算有查询、插入、删除、编辑、排序等。

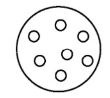
数据的逻辑结构通常有以下四种。

(1) 集合：结构中的数据元素之间除了仅仅同属于一个集合外，不存在其他关系，如图 1.4(a)所示。

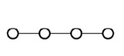
(2) 线性结构：数据元素之间存在一对一的关系，即元素之间有先后关系，如图 1.4(b)所示。

(3) 树形结构：数据元素之间存在一对多的关系，即元素之间有层次关系，如图 1.4(c)所示。

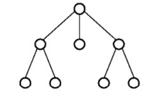
(4) 图状结构或网状结构：数据元素之间存在多对多的关系，即任意两个元素之间都可能有关系，如图 1.4(d)所示。



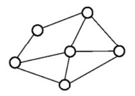
(a) 集合



(b) 线性结构



(c) 树形结构



(d) 图状或网状结构

图 1.4　逻辑结构

数据的存储结构可用以下四种基本的存储方法得到。

(1) 顺序存储方法。该方法是把逻辑上相邻的节点存储在物理位置上相邻的存储单元里，节点间的逻辑关系由存储单元的邻接关系来体现，得到的存储结构称为顺序存储结构，通常是借助于程序设计语言的数组来实现。例如，线性序列(a，b，c，d，e)的顺序存储结构如图 1.5 所示。数据的逻辑位序和其在数组中的下标(即物理位序)是一致的。

数组下标	元素	逻辑位序
0	*a*	1
1	*b*	2
2	*c*	3
3	*d*	4
4	*e*	5

图 1.5　顺序存储结构示意图

(2) 链式存储方法。

该方法是在每个数据元素中增加存放地址的指针域，通过指针来表示数据元素之间的逻辑关系。由此得到的存储结构称为链式存储结构，通常借助于程序设计语言的指针来实现。例如，线性序列(a，b，c，d，e)的链式存储结构如图 1.6 所示。显然，逻辑上相邻的元素在物理位置上不一定相邻。



物理位序	元素	指针	逻辑位序
2336	*a*	3046	1
3046	*b*	2000	2
2000	*c*	2140	3
2140	*d*	2378	4
2378	*e*	NULL	5

图 1.6　链式存储结构

一般地，链式存储结构经常表示为图 1.7 所示的链表形式。

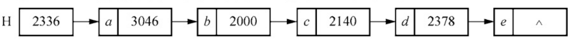


图 1.7　链表示意图

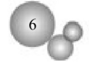
(3) 索引存储方法。该方法通常是在存储节点信息的同时，建立附加的索引表，根据索引表能快速地找到相应节点。索引表中的每一项称为索引项，索引项的一般形式是：(关键字，地址)，关键字是能唯一标识一个节点的数据项。

(4) 散列存储方法。该方法的基本思想是根据节点的关键字直接计算出该节点的存储地址。

上述四种基本的存储方法，既可以单独使用，又可以组合起来使用。同一种逻辑结构采用不同的存储方法，可以得到不同的存储结构。选择何种存储结构来表示相应的逻辑结构，视其具体要求而定，主要考虑运算方便及算法的时空要求。

5. 抽象数据类型

抽象数据类型(Abstract Data Type，ADT)是由一个数据结构以及定义在该数据结构上的



一组操作组成的。抽象数据类型的定义取决于数据的逻辑特性，与其在计算机内部的表示和实现无关。

抽象数据类型和数据类型实质上是同一个概念。例如，各种计算机都拥有的整数类型就是一个抽象数据类型，尽管它们在不同处理器上的实现方法可以不同，但由于其定义的数学特性相同，因此在用户看来都是相同的。因此，“抽象”的意义在于数据类型的数学抽象特性。

此外，抽象数据类型的含义更广，它不只局限于各种处理器中已定义并实现的数据类型，还包括用户在设计软件系统时自己定义的数据类型。在近代程序设计方法学中，要求在构成软件系统的每个相对独立的模块上，定义一组数据和施加在这些数据上的一组操作，在模块内部给出了这些数据的表示和操作的实现方法，而在模块外部使用的只是抽象的数据及其操作，这就是面向对象的程序设计方法。

1.3 算法和算法分析

因为数据的运算是通过算法描述的，所以讨论算法是“数据结构”课程的重要内容之一。

1.3.1 算法

通俗地讲，一个算法就是一种解题方法，是对特定问题求解步骤的一种描述。更严格地说，算法是由若干条指令组成的有限序列，它满足以下准则：

(1) 有穷性。一个算法必须总是在执行有穷步之后结束，且每一步都在有穷的时间内完成。

(2) 确定性。算法中每一条指令都必须有确切的含义，无二义性。

(3) 可行性。一个算法是可行的，是指算法描述的操作都是可以通过已经实现的基本运算执行有限次来实现的，即一个算法必须在有限的时间内完成。

(4) 输入。一个算法有零个或多个输入，作为算法加工的对象。

(5) 输出。一个算法有一个或多个输出，这些输出往往同输入有着某些特定的关系。

算法的含义与程序十分相似，但二者是有区别的。算法必须是有穷的，而一个程序不一定满足有穷性。例如，系统程序中的操作系统，只要整个系统不遭到破坏，它就永远不会停止，即使没有作业要处理，它也处于一个等待循环中，以等待新的作业进入，所以操作系统不是一个算法。因此，一个程序如果对任何输入都不会陷入无限循环，则它就是一个算法。另外，程序中的指令必须是机器可执行的，而算法中的指令则无此限制。因此，一个算法若能用机器可执行的语言来书写，则它就是一个程序。

一个算法可以用自然语言、数学语言或约定的符号语言来描述。本书将用 C 语言来描述算法。

1.3.2 算法分析

对于一个特定的实际问题，可以找出很多解决问题的算法。编程人员要想办法从中选择一个效率高的算法，这就需要有一个机制来评价算法。通常对一个算法的评价可以从算法执行的时间与算法所占用的内存空间两个方面来进行。内存空间一般可以通过增加计算

机的内存量来扩展，但是执行的时间是不可以扩展的，因此通常考虑时间要比考虑内存空间的情况多。本书主要讨论算法的时间特性，也会简单讨论空间特性。

一个算法所耗费的时间，是该算法中每条语句的执行时间之和，而每条语句的执行时间是该语句的执行次数(也称为频度)与该语句执行一次所需时间的乘积。但是，当算法转换为程序之后，每条语句执行一次所需的时间取决于机器执行指令的速度、编译所产生的代码质量等，这些是很难确定的，都与具体的机器有关，我们度量一个算法的效率应当抛开具体的机器，仅考虑算法本身的效率高低。因此我们假设执行每条语句所需的时间均是单位时间，则一个算法的时间耗费就是该算法中所有语句的频度之和。

例 1.1 求 $s=1+2+3+\cdots+n$，其算法如下。

```
long  sum(int  n)
{     int i;
      long s;
  (1) i=1;
  (2) while(i<n)
  (3) {   s=s+i;
  (4)    i++;
  (5) }
}
```

其中，语句(1)是赋值语句，只执行一次，故语句频度是 1。语句(2)的循环控制变量 i 从 1 增加到 n，当 $i\geqslant n$ 条件不成立时，才会终止，故它的频度是 n，但是它的循环体却只能执行 $n-1$ 次。语句(3)作为语句(2)的循环体执行 $n-1$ 次，而其自身(赋值语句)执行一次，因此语句(3)的频度是 $n-1$。同理，语句(4)的频度是 $n-1$。该算法的所有语句的频度之和(即算法的时间耗费)为

$$T(n)=1+n+n-1+n-1=3n-1 \tag{1.1}$$

由此可知，算法 sum 的时间耗费 $T(n)$是 n 的函数。

例 1.2 求下列算法段的语句频度：

```
(1) for(i=1; i<=n; i++)              n+1
(2)      for(j =1; j <=n ; j++)      n*(n+1)
(3)           x=x+1;                 n*n
```

其中，右边列出的是各语句的频度。语句(1)是循环语句，循环控制变量 i 从 1 增加到 $n+1$，当 $i\geqslant n+1$ 条件不成立时，循环终止，故它的频度是 $n+1$，但是它的循环体中每条语句均分别只执行了 n 次。语句(2)作为语句(1)的循环体内的语句执行 n 次，而其本身要执行 $n+1$ 次，所以语句(2)的频度是 $n(n+1)$。同理可得语句(3)的频度是 n^2。该算法段总的语句频度为

$$T(n)=(n+1)+n\times(n+1)+n\times n=2n^2+2n+1 \tag{1.2}$$

一般地，将算法求解问题的输入量称为问题的规模(或大小)，如上面例子中的 n。由上述例子可知，一个算法的时间频度 $T(n)$是该算法的时间耗费，它是该算法所求解问题规模 n 的函数。

当 n 不断变化时，时间频度 $T(n)$也会不断变化。但有时我们想知道它变化时呈现什么

规律。为此，我们引入时间复杂度的概念。

若有某个辅助函数$f(n)$，当n趋近于无穷大时，$T(n)/f(n)$的极限值为不等于零的常数，则称$f(n)$是$T(n)$的同数量级函数。

即 $\lim\limits_{n->\infty}\dfrac{T(n)}{f(n)}=c$，记作 $T(n)=O(f(n))$，则称 $O(f(n))$为算法的渐近时间复杂度，简称时间复杂度。例如，若 $T(n)=n(n+1)/2$，则有 $1/4\leqslant T(n)/n^2\leqslant 1$，故它的时间复杂度为 $O(n^2)$，即 $T(n)$与 n^2 数量级相同。

在例 1.1 中，算法的时间复杂度 $T(n)$如式(1.1)所示，当 n 趋向于无穷大时，有

$$\lim_{n->\infty}\frac{T(n)}{f(n)}=\lim_{n->\infty}\frac{3n-1}{n}=3$$

因此称该算法的渐进时间复杂度是 $T(n)=O(n)$。

同理，例 1.2 算法段的时间复杂度如式(1.2)所示，当 n 趋向于无穷大时，有

$$\lim_{n->\infty}\frac{T(n)}{f(n)}=\lim_{n->\infty}\frac{2\mathrm{n}^2+2\mathrm{n}+1}{n^2}=2$$

该算法的时间复杂度为 $T(n)=O(n^2)$。一般情况下，对循环语句只需考虑循环体中语句的执行次数，而忽略该语句中步长增减、终值判断、控制转移等成分。由此可见，当有若干个循环语句时，算法的时间复杂度是由嵌套层数最多的循环语句中最内层语句的频度$f(n)$决定的。

例 1.3 交换 x 和 y 的值。

```
temp=x;
x=y;
y=temp;
```

以上三条单个语句的频度均为 1，该程序段的执行时间是一个与问题规模 n 无关的常数，因此，算法的时间复杂度为常数阶，记作 $T(n)=O(1)$。事实上，只要算法的执行时间不随问题规模 n 的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数，此时，算法的时间复杂度也仅是 $O(1)$。

例 1.4 变量计数。

```
(1) for(i=1; i<=n; i++)
(2)       for(j =1; j <=i; j++)
(3)             x=x+1;
```

该程序段频度最大的语句是(3)，内循环变量 j 的执行次数虽然与问题规模 n 没有直接关系，但跟外层循环变量 i 有关，而 i 的执行次数跟 n 相关，因此 j 跟 n 也有关。因此语句(3)的执行次数可以从内层循环到外层分析得到：

$$\sum_{i=1}^{n}\sum_{j=1}^{i}1=\sum_{i=1}^{n}i=1+2+3+\cdots+n=n(n+1)/2$$

则该程序段的时间复杂度为 $T(n)=O(n(n+1)/2)=O(n^2)$。

在很多情况下，算法的时间复杂度不仅仅是问题规模 n 的函数，还与它所处理的数据集的状态有关。在这种情况下，通常会出现最好情况和最坏情况。我们经常对数据集的分

布做出某种假设(如等概率)，并讨论算法的平均时间复杂度。

通常，常见的时间复杂度，按数量级递增排列，依次为常数阶 $O(1)$、对数阶 $O(log_2n)$、线性阶 $O(n)$、线性对数阶 $O(nlog_2n)$、平方阶 $O(n^2)$、立方阶 $O(n^3)$、……、k 次方阶 $O(n^k)$、指数阶 $O(2^n)$。显然，时间复杂度为指数阶 $O(2^n)$的算法效率极低，当 n 稍大时就无法应用。

一个算法在计算机存储器上所占用的存储空间，包括存储算法本身所占用的存储空间、算法的输入输出数据所占用的存储空间和算法运行过程中临时占用的存储空间等。算法的空间复杂度(Space Complexity)主要是对一个算法运行过程中临时占用的存储空间大小的度量，记作

$$S(n)=O(f(n))$$

式中，n 为问题的规模。

渐进空间复杂度也常常简称为空间复杂度。

小　　结

本章从具体的应用例子，引出非数值处理问题的计算机处理方法及数据结构的由来；阐述了数据结构的基本概念和基本术语，包括数据、数据元素、数据类型、数据结构、抽象数据类型等；讨论了数据结构中常用的逻辑结构和存储结构；介绍了算法的概念及算法的度量和性能分析。

读者通过对本章的学习，能基本理解数据结构的基本概念和基本术语，能分析算法的时间复杂度。

习　　题

一、填空题

1．________是数据的基本单位，________是数据表示中最小的单位。

2．数据结构被形式地定义为$(D，R)$，其中 D 是________的有限集合，R 是 D 上的________的有限集合。

3．数据结构包括数据的________、数据的________和数据的________这三个方面的内容，狭义的数据结构就是指________。

4．数据结构按逻辑结构可分为四种，它们为________、________、________、和________。

5．线性结构中元素之间存在________关系，树形结构中元素之间存在________关系，图形结构中元素之间存在________关系。

6．数据的存储结构可用四种基本的存储方法表示，它们分别是________、________、________和________。

7．数据的运算最常用的有五种，它们分别是________、________、________、________和________。

8．算法是多条指令的________。

9．一个算法的效率可分为________效率和________效率。

二、选择题

1．非线性结构的数据元素之间存在(　　)。

A．一对多关系　　B．多对多关系

C．多对一关系　　D．一对一关系

2．数据结构中，与所使用的计算机无关的是数据的(　　)结构。

A．存储　　B．物理　　C．逻辑　　D．物理和存储

3．算法分析的目的是(　　)。

A．找出数据结构的合理性　　B．研究算法中的输入和输出的关系

C．分析算法的效率以求改进　　D．分析算法的易懂性和文档性

4．算法分析的两个主要方面是(　　)。

A．空间复杂性和时间复杂性　　B．正确性和简明性

C．可读性和文档性　　D．数据复杂性和程序复杂性

5．计算机算法指的是(　　)。

A．计算方法　　B．排序方法

C．解决问题的有限运算序列　　D．调度方法

6．计算机算法必须具备输入、输出和(　　)五个特性。

A．可行性、可移植性和可扩充性　　B．可行性、确定性和有穷性

C．确定性、有穷性和稳定性　　D．易读性、稳定性和安全性

三、简答题

1．数据结构和数据类型两个概念之间有区别吗？

2．简述程序与算法的异同点。

3．分析下面算法段中@语句的频度和算法的时间复杂度。

(1)

```
x=1,s=0;
for(i=1;i<=n;++i)
    {  ++x;
        s+=x;  ------@1
    }
```

(2)

```
x=1,s=0;
for(i=1;i<=n;++i)
    for(j=1;j<=n;++j)
    {  ++x;
        s+=x;  ------@2
    }
```

(3)

```
x=1,s=0;
while(x<=1000)
    {  ++x;
        s+=x;  ------@3
    }
```

(4)

```
x=1,s=0;
while(x<=n)
    {  ++x;
        s+=x;  ------@4
    }
```

(5)

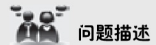
```
x=1,s=0;
do{  ++x;
    s+=x;  ------@5
  } while(x<=n)
```

(6)

```
x=0;
for(i=1;i<=n;i++){
    for(j=i;j<=n;j++)
        ++x;  ------@6
```



线　性　表



问题描述

学生成绩管理问题

学生成绩管理系统是学校教务部门日常工作的重要组成部分，其处理信息量很大。要求设计一个简单模拟的学生成绩管理系统，完成以下功能：输入学生数据(见表 2.1)，输出学生数据，学生信息查询(按学号查询、按姓名查询)，添加学生信息，修改学生数据，删除学生信息。

表 2.1　学生成绩信息

学号	姓名	专业班级	考试成绩
201105015101	吕心喜	计算机 111 班	91
201105015102	王飞	计算机 111 班	80
201105015103	宋晨霞	计算机 111 班	97
201105015104	陈伟杰	计算机 111 班	95
201105015105	张少英	计算机 111 班	79
201105015106	李国强	计算机 111 班	87

2.1　线性表的定义和基本操作

在这个学生成绩信息表中，每个学生的信息为一条记录，这条记录也称为数据元素。每个记录是一个节点，每个节点由学号、姓名、专业班级和考试成绩四个数据项组成。对于整个表来说，只有一个开始节点(它的前面无记录)和一个终端节点(它的后面无记录)，其他的节点则各有一个也只有一个直接前趋和直接后继(它的前面和后面均有且只有一个记录)。具有这种特点的逻辑结构称为线性表(线性结构)。

2.1.1　线性表的定义

1. 线性表的概念

线性表(Linear List)是由 $n(n\geqslant 0)$个数据元素(节点)a_1，a_2，…，a_n组成的有限序列。

其中：

(1) n 为数据元素的个数，也称表的长度。n=0 时，称为空表，记为( )。

(2) 常将非空的线性表($n>0$)，记作(a_1，a_2，…，a_n)。这里的数据元素 a_i($1\leqslant i\leqslant n$)只是一个抽象的符号，其具体含义在不同情况下可以不同。

数据元素类型多种多样，但同一线性表中的元素必定具有相同特性，即属于同一数据类型。表 2.2 中所有数据元素都为数字，表 2.3 中所有数据元素都为字符，表 2.4 中所有数据元素都为图片，而表 2.1 中所有数据元素都为记录(由若干数据项组成的数据元素)。

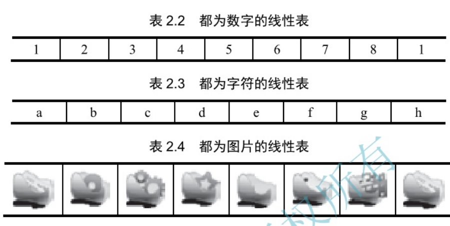
表 2.2　都为数字的线性表

1	2	3	4	5	6	7	8	1

表 2.3　都为字符的线性表

a	b	c	d	e	f	g	h

表 2.4　都为图片的线性表



2. 线性表的特点

数据元素的非空的线性表具有下面的特点：

(1) 有且仅有一个节点(a_1)没有直接前趋，称它为开始节点。

(2) 有且仅有一个节点(a_n)没有直接后继，称它为终端节点。

(3) 除开始节点外，线性表中其他任一节点 a_i($2\leqslant i\leqslant n$)都有且仅有一个直接前趋 a_{i-1}。

(4) 除终端节点外，线性表中其他任一节点 a_i($1\leqslant i\leqslant n-1$)都有且仅有一个直接后继 a_{i+1}。

2.1.2　识别线性表的基本操作

数据结构的基本操作是定义在逻辑结构层次上的，而这些操作的具体实现是建立在存储结构层次上的。在逻辑结构上定义的运算，只给出这些操作的功能是“做什么”，至于“如何做”等实现细节只有在确定了线性表的存储结构之后才能完成。

对于线性表的基本运算，常见的有以下几种：

(1) 初始化 InitList(&L)，即置空表，运算结果是将线性表 L 设置成一个空表。

(2) 求长度 Length(L)，当线性表 L 为非空时，返回表中的节点个数；当线性表为空时，返回 0。

(3) 判空表 Empty(L)，若线性表 L 为空表，则返回 TRUE，否则返回 FALSE。

(4) 取节点 GetElem(L，i)，当线性表 L 已存在($1\leqslant i\leqslant$Length(L))时，结果是返回 L 中第 i 个数据元素的值，否则返回 FALSE。

(5) 定位 Locate(L，item)，当线性表 L 中存在一个值为 item 的节点时，返回该节点的位置；当表 L 中存在多个值为 item 的节点时，返回首次找到的节点位置；当表 L 中不存在值为 item 的节点时，将给出一个特殊值表示值为 item 的节点不存在。

(6) 插入 Insert(&L，i，e)，在线性表 L 的第 i 个位置处插入一个值为 e 的新节点，使得原有表(a_1，…，a_{i-1}，a_i，a_{i+1}，…，a_n)变为表(a_1，…，a_{i-1}，e，a_i，a_{i+1}，…，a_n)。

(7) 删除 Delete(&L，i)，删除线性表 L 中的第 i 个节点，使得原有表(a_1，…，a_{i-1}，a_i，a_{i+1}，…，a_n)变为表(a_1，…，a_{i-1}，a_{i+1}，…，a_n)。

并非任何时候都需要同时执行以上运算。首先，不同问题中的线性表所需要执行的运算可能不同；其次，不可能也没有必要给出一组适合各种需要的运算，可以用基本运算的组合来实现。

例 2.1 利用线性表的基本运算实现清除表 *L* 中的重复节点。

实现该运算的基本思想：从表 *L* 的第一个节点(i=1)开始，逐个检查 i 位置以后的任一位置 j，若两节点相同，则将位置 j 上的节点从表 *L* 中删除，当遍历了 i 后面的所有位置之后，i 位置上的节点就成为当前表 *L* 中没有重复值的节点，然后将 i 向后移动一个位置。重复上述过程，直至 i 移动到当前表 *L* 的最后一个位置为止。该运算可用如下形式算法描述。

```
void Purge(Linear_list  L)              //删除线性表 L 中重复出现的节点
{ int i=1,j,x,y;
  while (i<Length(L))                   //每次循环都使第 i 个节点是无重复值的节点
    {  x=GetElem (L,i);
       j=i+1;
       while (j<=Length (L))
        {   y= GetElem (L,j);           //取当前第 j 个节点
            if (x==y)  Delete (&L,j);   //删除当前第 j 个节点
            else  j++;
        }
        i++;
    }
}           //Purge
```


算法中的 Delete 操作，使位置 j+1 上的节点及其后续节点均前移了一个位置，因此，应继续比较位置 j 上的节点是否与位置 i 上的节点相同；同时，Delete 操作使当前表长度减 1，故循环的终值分别使用了求长度运算 Length 以适应表长的变化。

请读者注意，这个算法对整型节点是正确的。若表 *L* 中元素是其他类型(即不是整型节点)，上述算法正确吗？若有误，应怎样修改？请读者思考。

2.2 线性表的顺序存储、实现和应用

如何将逻辑结构为线性表的学生成绩表存储到计算机中呢？这是存储结构的问题。数据结构在计算机中的表示称为存储结构。最常用的存储结构有顺序存储结构和链式存储结构，本节讨论顺序存储结构。

2.2.1 线性表的顺序存储结构

顺序存储结构是指把线性表的数据元素按逻辑次序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称顺序表。

在顺序表的存储结构中，假设表中每个节点占用 C 个存储单元，其中的第一个单元的存储地址则是该节点的存储地址，并设表中开始节点 a_1 的存储地址(简称为基地址)是 LOC(a_1)，那么节点 a_i 的存储地址 $LOC(a_i)$可以通过下式计算得到。

$$LOC(a_i)=LOC(a_1)+(i-1)\times C \qquad 1\leqslant i\leqslant n \tag{2.1}$$

顺序表的存储结构示意图如图 2.1 所示。

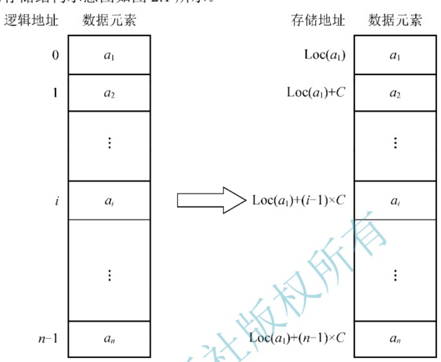


图 2.1 顺序表存储结构示意图

也就是说，在顺序表中，每个节点 a_i 的存储地址是该节点在表中的位置 i 的线性函数，只要知道基地址和每个节点的大小，就可在相同时间内求出任一节点的存储地址。因此顺序表是一种随机存取结构。

由于 C 语言中的向量(一维数组)也采用顺序存储表示，故可以用向量这种数据类型来描述顺序表。

```
typedef int ElemType;           // ElemType 可为任何类型,这里假设为 int
#define MAXSIZE 1024            //线性表可能的最大长度,这里假设为 1024
typedef struct
{ ElemType elem[MAXSIZE];       //线性表是向量存储,第一节点是 elem[0]
  int length;                   //定义 length 是线性表的长度
} SqList;
```

其中，数据域 elem 是存放线性表节点的向量空间，向量的下标从 0 到 MAXSIZE-1，线性表的第 i 个节点存放在向量的第 i-1 个分量中，下标是 i-1，并假设表中节点的个数始终不超过向量空间的大小 MAXSIZE；数据域 length 是线性表的长度；ElemType 是表中节点的数据类型，在此可认为它是某种定义过的类型，其含义视具体情况而定。例如，若线性表是整型数据，则 ElemType 就是标准类型 int；若线性表是英文字母表，则 ElemType 就是标准类型 char；若线性表是学生成绩表，则 ElemType 就是已定义过的表示学生成绩情况的结构类型。

当有以下定义时：

SqList　L；

其中，L 表示由一维数组 elem 和长度 length 组成的顺序表。L 中第 i 个节点表示为 L.elem[i-1]，L 的长度表示为 L.length。

而当 L 定义为：

SqList *L；时，L 为指向由一维数组 elem 和长度 length 组成的 SqList 顺序表类型的指针。L 中第 i 个节点表示为 L->elem[i-1](或者(*L).elem[i-1])，L 的长度表示为 L->length(或者(*L).length)。

总之，顺序表是用向量实现的线性表，向量的下标可以看作节点的相对地址。它的特点是逻辑上相邻的节点其物理位置也相邻。

2.2.2 顺序表的操作实现

定义了线性表的存储结构之后，就可以讨论在该存储结构上如何具体实现定义在逻辑结构上的运算了。

1. 顺序表的初始化

算法 2.1 构造一个空的顺序表。

构造一个空的顺序表，只要把表长置为 0 即可。

```
void InitList(SqList  *L)
{  L->length=0;     //空表,长度为 0
}
```

2. 定位(按值查找)

在顺序表中查找一个值为 item 的节点，当存在该节点时，返回该节点的位置；当表 L 中存在多个值为 item 的节点时，返回首次找到的节点位置；当表 L 中不存在值为 item 的节点时，返回 FALSE。

算法 2.2 在顺序表 L 中定位值为 item 的节点。

```
int Locate(SqList  L, ElemType item)
{  int i;
   for(i=0;i<L.length;i++)
      if(L.elem[i]= =item)
           return (i+1);
   printf("找不到该值!");
   return FALSE;
}
```

在本算法中，可能找到 item，此时平均比较次数为 $O(n/2)$，其中，n 是顺序表的长度；也有可能找不到 item，这时算法的比较次数为 $O(n)$，因此，本算法的平均时间复杂度为 $O(n)$。

3. 插入数据

线性表的插入运算是指在表的第 $i(1 \leqslant i \leqslant n+1)$个位置上，插入一个新节点 e，使长度为 n 的线性表(a_1，…，a_{i-1}，a_i，…，a_n)变成长度为 n+1 的线性表(a_1，…，a_{i-1}，e，a_i，…，a_n)。

用顺序表作为线性表的存储结构时，由于节点的物理顺序必须和节点的逻辑顺序保持

一致，因此我们必须将表中位置 n，$n-1$，…，i 上的节点后移到 $n+1$，n，…，$i+1$ 上，空出第 i 个位置，然后在该位置上插入新节点 e，仅当插入位置 $i=n+1$ 时，才无需移动节点，直接将 e 插入表的末尾。

其插入过程如图 2.2 所示。

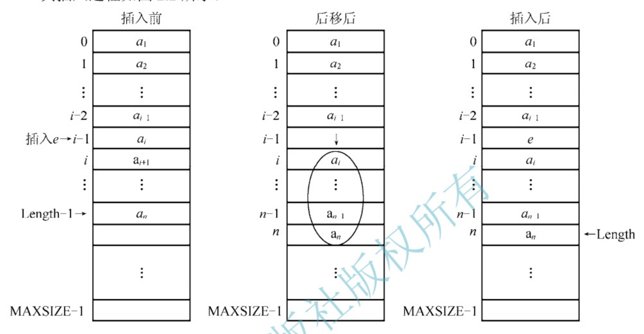


图 2.2　顺序表中插入节点 e 的过程

算法 2.3　插入数据算法描述如下。

```
int Insert(SqList *L, int i, ElemType e)
//将新节点 e 插入顺序表 L 的第 i 个位置上
{ int j;
if(L->length= = MAXSIZE)
    { printf("表满,溢出!");  return FALSE;  }
else if(i<1 || i>L->length)
    { printf("插入位置不合法!");  return FALSE;  }
else
    {   for (j=L->length-1; j>=i-1;j--)
            L->elem[j+1]= L->elem[j];         //节点后移
        L->elem[i-1]=e;                       //插入 e
        L->length++;                          //表长加 1
        return TRUE;
    }
}
```

注意：算法中节点后移的方向，必须从表中最后一个节点开始后移，直至将第 i 个节点后移为止。

分析算法的时间复杂度。该问题的规模是表的长度 L->length，设它的值为 n。显然该算法的时间主要花费在 for 循环中的节点后移语句上，该语句的执行次数(即移动节点的次数，即从 $a_i \cdots a_n$)是 $n-i+1$。由此可看出，所需移动节点的次数不仅依赖于表的长度 n，还与

插入位置 i 有关，当 $i=n+1$ 时，由于循环变量的终值大于初值，节点后移语句将不执行，无需移动节点；当 $i=1$ 时，则节点后移语句将循环执行 n 次，需移动表中所有节点。也就是说，该算法在最好情况下的时间复杂度是 $O(1)$；最坏情况下的时间复杂度是 $O(n)$。由于插入可能在表中任何位置上进行，因此，需分析算法的平均性能。

在长度为 n 的线性表中插入一个节点，令 $E_{\text{IS}}(n)$表示移动节点次数的期望值(即移动节点的平均次数)，在表中第 i 个位置上插入一个节点的移动次数为 $n-i+1$。故

$$E_{\text{IS}}=\sum_{i=1}^{n+1}p_i(n-i+1) \tag{2.2}$$

式中，p_i 表示在表中第 i 个位置上插入一个节点的概率。假设在表中任何合法位置($1\leqslant i\leqslant n+1$)上插入节点的机会是均等的，则

$$p_1=p_2=\cdots=p_{n+1}=1/(n+1)$$

因此，在等概率插入的情况下：

$$E_{\text{IS}}(n)=\sum_{i=1}^{n+1}(n-i+1)/(n+1)=n/2 \tag{2.3}$$

也就是说，在顺序表上做插入运算时，平均要移动表中的一半节点。当表长 n 较大时，算法的效率相当低。虽然 $E_{\text{IS}}(n)$ 中 n 的系数较小，但就数量级而言，它仍然是线性阶的，因此算法的平均时间复杂度是 $O(n)$。

4. 删除数据

线性表的删除运算是指将表的第 $i(1\leqslant i\leqslant n)$个节点删去，使长度为 n 的线性表(a_1，…，a_{i-1}，a_i，a_{i+1}，…，a_n)变成长度为 $n-1$ 的线性表(a_1，…，a_{i-1}，a_{i+1}，…，a_n)。

和插入运算类似，在顺序表上实现删除运算也必须移动节点，才能反映出节点间逻辑关系的变化。若 i=n，则只要简单地删除终端节点，无需移动节点；若 $1\leqslant i\leqslant n-1$，则必须将表中节点 a_{i+1}，a_{i+2}，…，a_n，依次前移到位置 i，$i+1$，…，$n-1$ 上，将原有位置上的节点覆盖，以实现删除。其删除过程如图 2.3 所示。

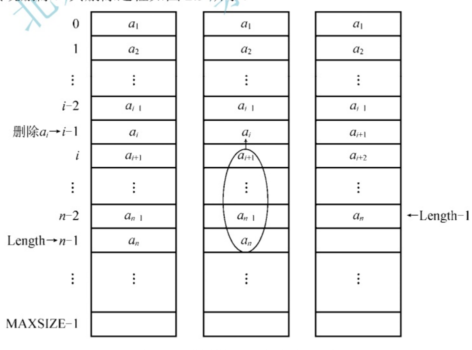


图 2.3 顺序表中删除节点的过程

算法 2.4 删除节点算法描述如下。

```
int Delete(SqList *L,int i)                    //从顺序表中删除第 i 个位置上的节点
{   int j;
    if(L->length==0)
       { printf("表为空,无法删除!");  return FALSE;  }
    else if(i<1 || i>L->length)
       { printf("删除位置不合法!");  return FALSE;  }
    else
       { for (j=i; j<= L->length-1;j++)
            L->elem[j-1]= L->elem [j];  //节点前移
         L->length--;                   //表长减 1
         return TRUE;
       }
}
```

该算法的时间分析与插入算法类似，节点的移动次数也是由表长 n 和位置决定的。若 $i=n$，则由于循环变量的初值大于终值，前移语句将不执行，无需移动节点；若 $i=1$，则前移语句将循环执行 $n-1$ 次，需移动表中除开始节点外的所有节点。这两种情况下算法的时间复杂度分别是 $O(1)$和 $O(n)$。

删除算法的平均性能分析与插入算法相似。在长度为 n 的线性表中删除一个节点，令 $E_{\mathrm{DE}}(n)$ 表示所需移动节点的平均次数，删除表中第 i 个节点的移动次数为 $n-i$，故

$$E_{\mathrm{DE}}(n)=\sum_{i=1}^{n}p_i(n-i) \tag{2.4}$$

式中，p_i 表示删除表中第 i 个节点的概率。在等概率的假设下，$p_1=p_2=\cdots=p_n=1/n$，由此可得：

$$E_{\mathrm{DE}}(n)=\sum_{i=1}^{n}(n-i)/n=(n-1)/2 \tag{2.5}$$

即在顺序表上做删除运算，平均要移动表中约一半的节点，平均时间复杂度也是 $O(n)$。

2.2.3 用顺序表实现学生成绩管理问题

(1) 定义学生成绩表中数据元素类型。

```
typedef struct                       //定义学生成绩表中数据元素的类型
{   char num[13];                    //学号
    char name[8];                    //姓名
    char classname[15];              //专业班级
    int score;                       //成绩
} ElemType;                          //学生信息

typedef struct
{  ElemType elem[MAXSIZE];           //存储各学生信息的数组
    int length;                      //学生数
}SqList;                             //顺序表类型
```

(2) 编写程序，实现学生成绩管理系统中要求的各项功能。

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 100
#define TRUE 1
#define FALSE 0

typedef struct              /*定义学生成绩表中数据元素类型*/
{   char num[13];           /*学号*/
    char name[8];           /*姓名*/
    char classname[15];     /*专业班级*/
    int score;              /*成绩*/
} ElemType;                 /*学生信息*/
typedef struct
{ElemType elem[MAXSIZE]; /*存储各学生信息的数组*/
 int length;                /*学生数*/
}SqList;                    /*顺序表,存储学生列表*/

void InitList(SqList  *L)
/*初始化顺序表*/
{  L->length=0;             /*空表,长度为 0*/
}

void CreateSeqList(SqList *L)
/*从文件里读取数据,构建顺序表*/
{   FILE *fp;
    int i=0;
    fp=fopen("D:\\SqListAPP\\StudentInfo.txt","r"); /*数据文件所在的路径,
即当前数据文件 StudentInfo.txt 在 D 盘的 SqListAPP 目录下,请读者根据自己实际存放的路径进
行修改,下同*/
    if(!fp)
        printf("\nCan not open file!\n");    /*数据文件无法打开*/
    else
        {
         while(!feof(fp))
         {   fscanf(fp,"%s %s %s %d",L->elem[i].num,
                      L->elem[i].name,L->elem[i].classname,&(L->elem[i].
score));
             L->length++;
             i++;
        }
        L->length--;            /*L->length 减 1 后才是顺序表的真正长度*/
        }
}

void FWrite(SqList L)          /*把顺序表中的数据写回到文件里*/
{   FILE *fp;
    int i=0;
    fp=fopen("D:\\SqListAPP\\StudentInfo.txt","w");
    if(!fp)
```

```
        printf("\nCan not open file!\n");   /*数据文件无法打开*/
    else
        {
         while(i<L.length)
         {   fprintf(fp,"%s %s %s %d\n",L.elem[i].num,L.elem[i].name,
                    L.elem[i].classname,L.elem[i].score);
             i++;
         }
        }
}

void OutPut(SqList L)   /*输出顺序表*/
{   int i;
    printf("\n 学号\t\t 姓名\t 专业班级\t 成绩 \n");
    for(i=0;i<L.length-1;i++)
      printf("%s\t%s\t%s\t%d\n",L.elem[i].num,
              L.elem[i].name,L.elem[i].classname,L.elem[i].score);
    printf("\n 数据输出完毕!");
}

int Insert(SqList *L, int i, ElemType x)/*将新结点 x 插入顺序表 L 的第 i 个位置上*/
{ int j;
   if(L->length==MAXSIZE)
      { printf("表满,溢出! ");  return FALSE;  }
   else if(i<1 || i>L->length)
             { printf("插入位置不合法! ");  return FALSE;  }
            else
             { for (j=L->length-1; j>=i-1;j--)
                L->elem[j+1]= L->elem[j];    /*结点后移*/
               L->elem[i-1]=x;      /*插入 x */
               L->length++;     /*表长加 1*/
               return TRUE;
            }
}

int Delete(SqList *L,int i)           /*从顺序表中删除第 i 个位置上的结点*/
{  int j;
   if(L->length==0)
    { printf("表为空,无法删除! ");  return FALSE;  }
   else if(i<1 || i>L->length)
        { printf("删除位置不合法! ");  return FALSE;  }
       else
         { for (j=i; j<= L->length-1;j++)
              L->elem[j-1]= L->elem [j];  /*结点前移*/
           L->length - -;                 /*表长减 1*/
           return TRUE;
         }
}

int Locate(SqList L, char number[12])  /*在顺序表中查找一个值为 number 的结点*/
{  int i;
   for(i=0;i<L.length;i++)
      if(strcmp(L.elem[i].num,number)==0)
```

```
                return (i+1);
          if(i>=L.length)
            { printf("找不到该值! "); return FALSE; }
       }

       int main( )
       {
        SqList Student;
        ElemType e;
        char number[13],yn;
        int n,loc;
        InitList(&Student);
        CreateSeqList(&Student);
        while(1)
         { fflush(stdin);
           system("cls");
           printf("\n");
           printf("******** 学生成绩管理系统 *******\n");
           printf("*      1-输出所有学生信息       *\n");
           printf("*      2-按学号查询学生信息     *\n");
           printf("*      3-修改学生数据           *\n");
           printf("*      4-添加一个学生信息       *\n");
           printf("*      5-删除一个学生信息       *\n");
           printf("*      0-退出系统               *\n");
           printf("*********************************\n");
           printf("请选择(Select):");
           switch(getche())
           {
            case '1':    /*输出所有学生信息*/
                     OutPut(Student);
                     break;
            case '2':    /*按学号查询学生信息*/
                     printf("\n 请输入待查询学生的学号:");
                     gets(number);
                     loc=Locate(Student,number);
                     if(loc)
                     {
                         printf("\n 找到,该学生信息为:");
                         printf("\n   学  号\t 姓 名\t 专业班级\t 成  绩 \n");
                         printf("%s\t%s\t%s\t%d\n",Student.elem[loc-1].num,
                                Student.elem[loc-1].name,Student.elem[loc-1].
classname,
                                Student.elem[loc-1].score);
                     }
                     else  printf("\n 该学号的学生信息不存在! \n");
                     break;
             case '3':   /*修改学生数据*/
                     printf("\n 请输入待修改学生的学号:");
                     gets(number);
                     loc=Locate(Student,number);
                     if(!loc)
                         printf("\n 该学号的学生信息不存在!");
                     else
```

```
                        {
                            printf("\n 找到,该学生信息为:");
                            printf("\n 学号\t\t 姓名\t 专业班级\t 成绩 \n");
                            printf("%s\t%s\t%s\t%d\n",Student.elem[loc-1].num,
                                Student.elem[loc-1].name,Student.elem[loc-1].
classname,
                                Student.elem[loc-1].score);
                            printf("\n 修改学生信息为: ");
                            printf("\n 学号: "); scanf("%s",Student.elem[loc-1].num);
                            printf("\n 姓名: "); scanf("%s",Student.elem[loc-1].name);
                            printf("\n  专 业 :  ");scanf("%s",Student.elem[loc-1].
classname);
                            printf("\n 成绩: "); scanf("%d",&Student.elem[loc-1].score);
                            FWrite(Student);
                            puts("\n 修改数据成功!");
                        }
                        break;
        case '4':       /*添加一个学生信息*/
                        printf("\n 请输入待插入学生的信息: ");
                        printf("\n 学号: "); scanf("%s",e.num);
                        printf("\n 姓名: "); scanf("%s",e.name);
                        printf("\n 专业: "); scanf("%s",e.classname);
                        printf("\n 成绩: "); scanf("%d",&e.score);
                        printf("\n 请输入插入位置(1~%d): ",Student.length);
                        scanf("%d",&loc);
                        Insert(&Student, loc, e);
                        FWrite(Student);
                        puts("\n 数据添加成功!");
                        break;
        case '5':       /*删除一个学生信息*/
                        printf("\n 请输入待删除学生的学号: ");
                        gets(number);
                        loc=Locate(Student,number);
                        if(loc)
                        {
                            printf("\n 找到,该学生信息为: ");
                            printf("\n 学号\t\t 姓名\t 专业班级\t 成绩 \n");
                            printf("%s\t%s\t%s\t%d\n",Student.elem[loc-1].num,
                                   Student.elem[loc-1].name,Student.elem[loc-1].
classname,
                                   Student.elem[loc-1].score);
                            printf("\n 确定是否删除: (Y/N)");
                            yn=getchar( );
                            if((yn=='Y'||yn=='y'))
                            {
                             Delete(&Student, loc);
                             FWrite(Student);
                             puts("\n 数据删除成功!");
                            }
                        }
                        else  printf("\n 该学号的学生信息不存在! ");
                        break;
         case '0':      /*退出系统*/
```

```
                printf("\n 感谢使用本学生成绩管理系统！ \n");
                return 0;
            }
            puts("\n 按任意键返回！ \n");
            getche( );
        }
    }
```

上述程序首先将表 2.1 的学生成绩信息保存在文件 StudentInfo.txt 中，其次用 InitList(&Student)函数初始化顺序表 Student，再用 CreateSeqList(&Student)函数将数据从文件读入顺序表 Student，最后才能进行查询、修改、插入、删除等操作。



独立实践

(1) 按姓名查询学生成绩信息，并考虑能否查询多个相同姓名学生的信息。

(2) 修改上述程序中的“3-修改学生数据”功能，使其能保证输入的学号唯一。

2.3 线性表的链式存储、实现和应用

前面研究了线性表的顺序存储结构，其特点是用物理位置上的邻接关系来表示节点间的逻辑关系，这一特点使得顺序存储结构有如下的优缺点。

其优点如下。

(1) 无需为表示节点间的逻辑关系而增加额外的存储空间。

(2) 可以方便地随机存取表中任一节点。

其缺点如下。

(1) 插入或删除运算不方便，除表尾的位置外，在表的其他位置上进行插入或删除操作时都必须移动大量的节点，其效率较低。

(2) 由于顺序表要求占用连续的存储空间，存储分配只能预先进行(静态分配)。因此，当表长变化较大时，难以确定合适的存储规模。若按可能达到的最大长度预先分配表空间，则可能造成一部分空间长期空置而得不到充分利用；若事先对表长估计不足，则插入操作可能使表长超过预先分配的空间而造成溢出。

为了克服顺序表的缺点，可以采用链接方式存储线性表，通常我们将链接方式存储的线性表称为链表(Linked List)。它不要求逻辑上相邻的元素在物理位置上也相邻，因此它没有顺序存储结构所具有的缺点。按照指针域的组织及各个节点之间的联系形式，链表又可以分为单链表、循环链表、双链表等多种类型。



问题描述

病患信息管理问题

病患信息系统需要把病患的基本信息登录进系统内，然后对其进行增加、删除、查询、修改等相关操作，这就需要把基本的信息在计算机中存储下来，因病患人数的最大值难以确定，故用动态的处理较节省空间，则往往会用链表的形式来存储这些数据。

2.3.1 单链表

1. 单链表的基本结构

链表用一组任意的存储单元来存储线性表中的数据元素，这组存储单元可以是连续的，也可以是不连续的，甚至可以零散分布在内存中的任何位置上。那么，怎么表示两个数据元素逻辑上的相邻关系，即如何表示数据元素之间的线性关系呢？为此，在存储数据元素时，除了存储数据元素本身的信息外，还必须存储指示其后继节点的地址(或位置)信息，这个信息称为指针(Pointer)或链(Link)。这两部分信息组成了链表中的节点结构，如图 2.4 所示。

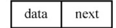


图 2.4　单链表的节点结构

其中，data 域是数据域，用来存放节点的值；next 域是指针域(亦称链域)，用来存放节点的直接后继的地址(或位置)。链表正是通过每个节点的链域将线性表的 n 个节点按其逻辑顺序链接在一起的。由于上述链表的每个节点只有一个链域，故将这种链表称为单链表(Single Linked List)。

假设有一个线性表{ZHAO，QIAN，SUN，LI，ZHOU，WU，ZHENG，WANG}，用单链表存储的内存示意图如图 2.5 所示，从图中可以看出，逻辑相邻的两个字符串如“ZHAO”与“QIAN”的存储空间是不连续的，通过在“ZHAO”的指针域存放“QIAN”的存储位置 7 来表示两者逻辑上的邻接关系。另外，“WANG”的后面没有其他元素，因此，它的指针域值为 NULL。



存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25

图 2.5　单链表的内存示意图

为方便表示，往往将图 2.5 简化为图 2.6 表示的单链表。

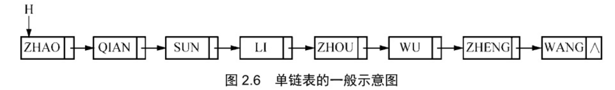


图 2.6　单链表的一般示意图

单链表是最简单的链表，它由头指针唯一确定，因此单链表可以用头指针的名称来命名，例如，若头指针名是 H，则把链表称为表 H。表中每个节点指向列表中的下一个节点，最后一个节点不指向任何其他节点，而指向 NULL，代表链表结束。如果有 H=NULL，则表示该单链表是一个空表，长度为 0。

用 C 语言描述单链表如下。

```
typedef int ElemType;
typedef struct LNode      //节点类型定义
 {ElemType data;
   struct LNode *next;
 } LNode, *LinkList;
```

值得一提的是，我们一定要严格区分指针变量和节点变量这两个概念。

```
LNode   x;                  // x是一个结构体节点变量
LinkList  L, p;             //L、p是结构体指针变量
```

例如，如上定义的变量 x 是结构体 LNode 类型的节点变量，而 p 是类型为 LinkList 的指针变量，也可以用结构体 LNode 类型来定义指针 L，p。

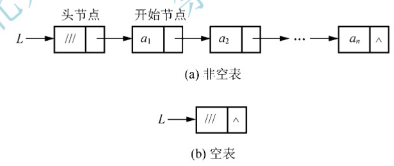
```
LNode   *L,*p;              //相当于LinkList  L,p;
```

通常 p 所指的节点变量并非在变量说明部分明显地定义，而是在程序执行过程中，当需要时才产生，故称为动态变量。实际上，它是通过标准函数生成的，即

```
          p=(LinkList)malloc(sizeof(LNode));
```

函数 malloc 分配一个类型为 LNode 的节点变量的空间，并将其地址放入指针变量 p 中。一旦 p 所指向的节点变量不再需要了，又可通过标准函数 free(p)；释放 p 所指的节点变量空间。

有时，为了更方便地判断空表、插入和删除等操作，使空表和非空表的处理一致，在单链表的第一个节点前面加上一个附设的节点，称为头节点。图 2.7 所示即为带头节点的单链表，单链表的头指针 L 指向头节点。如果头节点的指针域为空，即 L->next 等于 NULL，则表示该链表为空表。



(a) 非空表

(b) 空表

图 2.7　带头节点的单链表

2. 单链表的操作实现

下面讨论用带头节点的单链表做存储结构时，如何实现线性表的几种基本运算。

1) 建立单链表

假设线性表中节点的数据类型是整型，建立具有 n 个元素的单链表，元素从键盘上输入。动态地建立单链表的常用方法有以下两种。

(1) 尾插法建表。

元素从键盘上输入，输入-999 表示结束。

① 产生一个新节点，由头指针 L 指向该节点，并使指针 p 也指向该节点，如图 2.8(a)所示。实现语句：L=(LinkList) malloc(sizeof(LNode))；p=L；

② 从键盘上输入一个整型元素 x，当 x 为−999 时，转到步骤⑤；当 x 不等于−999 时，产生一个新节点，用 p->next 指针指向该新节点，如图 2.8(b)所示。实现语句：p->next =(LinkList) malloc(sizeof(LNode))；

③ 将指针变量 p 移动到 p->next 指针指向的节点，语句为 p=p->next，并将 x 放入该节点数据域的值，语句为 p->data=x，如图 2.8(c)所示。

④ 从键盘上输入一个元素 x，重复步骤②、③。

⑤ 当键盘上输入 x 的值为−999 时循环结束，到此已建成了如图 2.8(d)所示的单链表。此时，最后终端节点的指针域没有值，将其置为 NULL，语句：p->next=NULL。

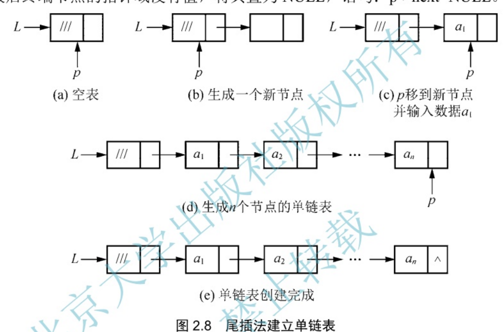


图 2.8 尾插法建立单链表

于是，一个带头节点有 n 个元素的单链表建立完毕。

算法 2.5 尾插法建立单链表。

```
void CreateListR(LinkList  L)
{ int x;   LinkList  p;
  L=(LinkList)malloc(sizeof(LNode));              //分配一个节点作为头节点
  p=L;
  scanf("%d",&x);
  while(x!=-999)
  { p->next=(LinkList)malloc(sizeof(LNode));      //为后继节点创建新空间
    p=p->next;                //指针 p 移向新节点
    p->data=x;                //把 x 放入 p 的数据域中
    scanf("%d",&x);
  }
  p->next=NULL;               //单链表尾节点的指针域置空
}
```

尾插法建表比较方便，建成的表节点顺序与数据输入的顺序一致，因此，它是一种最为常用的建表方法。有时，需要建成的表节点顺序与数据输入的顺序相反，这时用头插法建表就能实现。

(2) 头插法建表。

该方法从一个空表开始，重复读入数据，生成新节点，将读入的数据存放到新节点的数据域中，然后将新节点插入到当前链表的表头上，直至读入结束标志为止。

过程描述如图 2.9 所示。

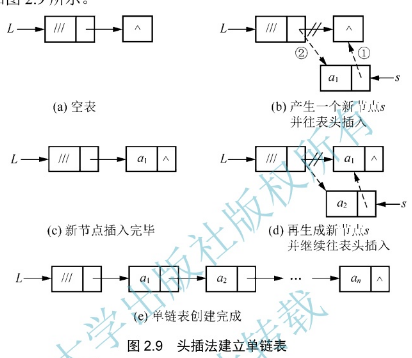


图 2.9 头插法建立单链表

算法 2.6 头插法建立单链表。

```
void CreateListF(LinkList L)
{ int x;  LinkList  s;
  L=(LinkList)malloc(sizeof(LNode));
  L->next=NULL;                               //先建立一个带头节点的空单链表
  scanf("%d",&x);
while(x!=-999)
  { s=(LinkList)malloc(sizeof(LNode));        //生成新节点
    s->data=x;                                //将 x 放入 s 的数据域
    s->next=L->next;  L->next=s;              //插入到表头
    scanf("%d",&x);                           //继续输入元素 x
  }
}
```

以上两个算法的时间复杂度均是 $O(n)$。

2) 定位(按值查找)

定位(按值查找)是在链表中，查找是否有节点值等于给定值 item 的节点，若有，则返回首次找到的其值为 item 的节点的地址；否则返回 NULL。查找过程从开始节点出发，顺着链表逐个将节点的值和给定值 item 做比较，直到找到 item，即查找成功；或者已经到链表结束，查找失败，如图 2.10 所示。

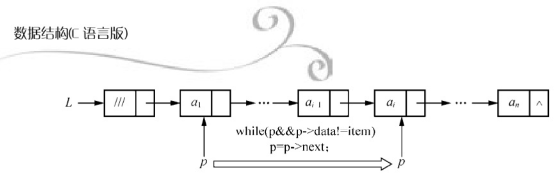


图 2.10　在单链表中查找值为 item 的节点

算法 2.7　按值查找。

```
LinkList Locate(LinkList L,ElemType item)
//在链表中查找一个值为 item 的节点
{ LinkList p;
 p=L->next;
 while(p&&p->data!=item)
    p=p->next;
 if(!p) return NULL;
 else return(p);
}
```

在有些应用中，按值查找时需返回找到的 item 的位置，请读者根据上述算法自行编写。

在算法 2.7 中，while 语句的终止条件是搜索到表尾或者满足 p->data==item，它和被寻找元素的所在位置有关。总之，在第 i 个位置找到时最多需要比较次数为 i 次。因而，在等概率假设下，平均时间复杂度为：

$$\sum_{i=1}^{n} i/n = 1/n \cdot \sum_{i=1}^{n} i = (n+1)/2 = O(n)$$

3) 插入数据

假设指针 p 指向单链表的某一节点，指针 s 指向待插入的、其值为 x 的新节点。若将新节点*s 插入节点*p 之后，则简称为“后插”；若将*s 插入在*p 之前，则简称为“前插”。两种插入操作都必须先生成新节点，然后修改相应的指针，再插入。这里以后插为例，前插请读者自行完成。

后插操作较简单，其插入过程如图 2.11 所示。其算法如下。

```
void InsertAfter(LinkList p, ElemType x)    //将值为 x 的新节点插入*p 之后
{ LinkList s;
  s=(LinkList)malloc(sizeof(LNode));        //生成新节点*s,图中步骤①
  s->data=x;                                //图中步骤②
  s->next=p->next;                          //图中步骤③
  p->next=s;                                //将*s 插入*p 之后,图中步骤④
}
```

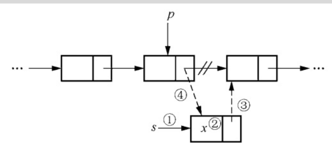


图 2.11　在节点 p 之后插入节点 s

单链表的插入运算 Insert(L, i, x)是生成一个值为 x 的新节点，并将其插入到链表 L 中第 i 个节点之前，也就是插入到第 i-1 个节点之后。步骤：首先从头节点开始找到第 i-1 个节点，用指针 p 指向，然后生成一个新节点 s，将值 x 放入 s 的数据域，再将 s 插入到 p 后，完成该运算。过程如图 2.12 所示。

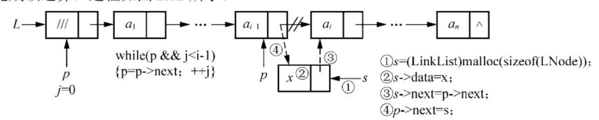


图 2.12　插入过程示意图

算法 2.8　插入运算。

```
 int Insert(LinkList  L, int i, ElemType item)
//在单链表上 L 的第 i 个节点前插入元素 item
{ LinkList  p,s; p=L;
  int j=0;
  while(p && j<i-1)   //找到第 i-1 个节点
     {  p=p->next;  ++j; }
  if( !p || j>i-1) return FALSE;
  s=(LinkList)malloc(sizeof(LNode));
  s->data= item;
  s->next=p->next; p->next=s;
  return TRUE;
}
```


设单链表的长度为 n，合法的前插位置是 $1 \leqslant i \leqslant n+1$，算法 Insert 的时间主要耗费在查找操作上，所以时间复杂度为 $O(n)$。

4) 删除数据

和插入运算类似，要删除单链表中节点*p 的后继很简单，首先用一个指针 r 指向被删节点，然后修改*p 的指针域，最后释放节点*r。其过程如图 2.13 所示，图中的“存储池”是备用的节点空间，释放节点就是将节点空间归还到存储池中。

该过程可用语句描述如下。

```
        LinkList r;
        r=p->next;               //示意图中步骤①
        p->next=r->next;         //图中步骤②,将节点*r 从链表上删除
        free(r);                 //图中步骤③,释放节点
```

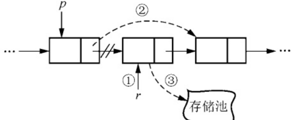


图 2.13　删除*p 之后的节点

若被删节点就是 p 所指的节点本身，则和前插问题类似，必须修改*p 的前趋节点*q 的指针域。因此一般情况下也要从头指针开始顺着链表找到*p 的前趋节点*q，然后删除*p。其删除过程如图 2.14 所示。

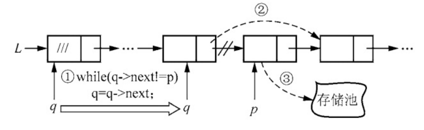


图 2.14 删除节点*p

算法描述如下。

```
int DeleteP(LinkList L,LinkList p)
//在单链表 L 中删除 p 节点本身
{LinkList q;
 q=L;                              //q 指向头节点,从头开始查找
 while(q->next && q->next!=p)      //找到*p 的前趋*q
      q=q->next;
 if(q->next==NULL)  return FALSE;
 else
 {    q->next=p->next;             //从链表中删除*p
      free(p);                     //释放空间,将其归还存储池
      return TRUE;
 }
}
```


另外，还有一种较为简单的方法，即把*p 节点的后继节点的值前移到*p 节点中，然后删除*p 的后继。此法要求*p 有后继，不是终端节点。请读者自己完成删除节点*p 的算法。

单链表的删除运算 Delete(L，i)是指删除单链表中第 i 个节点，方法类似于插入运算：从头节点开始找到第 i−1 个节点，再删除其后继节点。删除过程如图 2.15 所示。

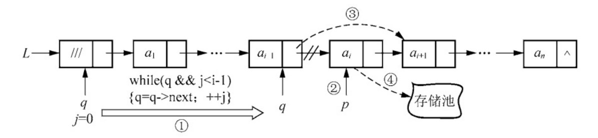


图 2.15　删除过程示意图

算法 2.9　删除运算。

```
int Delete (LinkList L, int i)
//删除单链表 L 中第 i 个节点
{ LinkList p,q;
int j;
```

```
        q=L; j=0;
        while(q && j<i-1)
         { q=q->next;  ++j;}
        if( !q || j>i-1) return FALSE;
        p=q->next;
        q->next=p->next;
        free(p);
        return TRUE;
    }
```

设单链表的长度为 n，则删除第 i 个节点仅当 $1 \leqslant i \leqslant n$ 时是合法的。显然算法 Delete 的时间复杂度也是 $O(n)$。

从上面的讨论可以看出，链表上实现的插入和删除运算，无需移动节点，仅需修改指针。

2.3.2 用单链表实现病患信息管理问题

下面用带头节点的单链表做存储结构，来实现病患信息管理系统。

(1) 定义病患信息数据元素类型。

```
typedef struct
{              //病患信息节点类型
    char num[5];            //编号
    char name[9];           //姓名
    char sex[3];            //性别
    char phone[13];         //电话
    char addr[31];          //地址
}ElemType;

typedef struct node
{    //节点类型定义
    ElemType data;          //节点数据域
    struct node * next;     //节点指针域
}LNode, *LinkList;
```

(2) 编写程序，实现病患信息管理系统中要求的各项功能。

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

LinkList pHead;
LNode *p;

//函数说明
```

```
int menu_select();
LinkList CreateList(void);
void InsertNode(LinkList head,LNode *p);
LNode *ListFind(LinkList head);
void DeleteNode(LinkList head);
void PrintList(LinkList head);

//主函数
void main()
{
    for(;;)
    {
        switch(menu_select())
        {
        case 1:
            printf("*****************************************\n");
            printf("*          病患信息链表的建立(Tab 键分隔) *\n");
            printf("*****************************************\n");
            pHead=CreateList();
            break;
        case 2:
            printf("*****************************************\n");
            printf("*           病患信息的添加           *\n");
            printf("*****************************************\n");
            printf("*编号(4) 姓名(8) 性别 电话(11) 地址(31)*\n");
            printf("*****************************************\n");
            p=(LNode *)malloc(sizeof(LNode));    //申请新节点
            scanf("%s%s%s%s%s",p->data.num,p->data.name,
                p->data.sex,p->data.phone,p->data.addr);
            InsertNode(pHead,p);
            break;
        case 3:
            printf("*****************************************\n");
            printf("*          病患信息的查询            *\n");
            printf("*****************************************\n");
            p=ListFind(pHead);
            if(p!=NULL)
            {
                printf("*编 号  姓 名  性 别  联系电话  地 址  *\n");
                printf("----------------------------------------\n");
                printf("%s\t%s\t%s\t%s\t%s\n",p->data.num,p->data.name,
                    p->data.sex,p->data.phone,p->data.addr);
                printf("----------------------------------------\n");
```

```
            }
            else
                printf("没有查到要查询的病人! \n");
            break;
        case 4:
            printf("******************************************\n");
            printf("*           病患信息的删除             *\n");
            printf("******************************************\n");
            DeleteNode(pHead);     //删除节点
            break;
        case 5:
            printf("******************************************\n");
            printf("*           病患信息链表的输出           *\n");
            printf("******************************************\n");
            PrintList(pHead);
            break;
        case 0:
            printf("\t 再见!\n");
            return;
        }
    }
}

int menu_select()        //菜单选择函数程序
{
    int sn;
    printf("===============================\n");
    printf("   病患信息管理系统\n");
    printf("===============================\n");
    printf("   1.病患信息链表的建立\n");
    printf("   2.病患节点的插入\n");
    printf("   3.病患节点的查询\n");
    printf("   4.病患节点的删除\n");
    printf("   5.病患信息链表的输出\n");
    printf("   0.退出管理系统\n");
    printf("===============================\n");
    for(;;)
    {
        scanf(" %d",&sn);
        if(sn<0||sn>5)
            printf("\n\t 输入错误,重选 0-5");
        else
            break;
```

```
    }
    return sn;
}

LinkList CreateList(void)          //尾插法建立带头节点的病患信息链表算法
{
    LinkList head = (LNode*) malloc(sizeof(LNode)); //申请表头节点
    LNode *p, *rear;
    int flag = 0;                  //结束标志置 0
    rear = head;                   //尾指针初始指向头节点
    while(flag == 0)
    {
        p=(LNode*)malloc(sizeof(LNode));      //申请新节点
        printf("编号(4)\t 姓名(8)\t 性别\t 电话(11)\t 地址(31)  \n");
        printf("-----------------------------------------\n");
        scanf(" %s\t%s\t%s\t%s\t%s", p->data.num, p->data.name, p->data.sex,
            p->data.phone, p->data.addr);
        rear->next = p;            //新节点连接到尾节点之后
        rear=p;                    //尾指针指向新节点
        printf("结束建表吗?  (1/0):");
        scanf(" %d",&flag);        //读入一个标志数据
    }
    rear->next=NULL;               //终端节点指针域置空
    return head;
}
void InsertNode(LinkList head,LNode *p)      //在病患信息链表head中插入节点
{
    LNode *p1,*p2;
    p1 = head;
    p2 = p1->next;
    while(p2 != NULL && strcmp(p2->data.num, p->data.num) < 0)
    {
        p1 = p2;                   //p1 指向刚访问过的节点
        p2 = p2->next;             //p2 指向表的下一个节点
    }
    p1->next = p;                  //插入 p 所指向的节点
    p->next = p2;                  //连接表中剩余部分
}

LNode * ListFind(LinkList head)  //有序病患信息链表上的查找
{
    LNode *p;
    char num[5];
```

```
    char name[9];
    int xz;
    printf("=================\n");
    printf("  1.按编号查询    \n");
    printf("  2.按姓名查询    \n");
    printf("=================\n");
    printf("请选择:");
    p = head->next;                     //假设病患信息表带头节点
    scanf("%d", &xz);
    fflush(stdin);
    if(xz==1)
    {
        printf("请输入要查找者的编号: ");
        scanf("%s", num);
        fflush(stdin);
        while(p&&strcmp(p->data.num, num) < 0)
         p = p->next;
        if(p == NULL || strcmpi(p->data.num, num) > 0)
            p = NULL;                   //没有查到要查找的病患
    }
    else
    {
        if(xz == 2)
            printf("请输入要查找者的姓名: ");
        scanf("%s", name);
        fflush(stdin);
        while(p && strcmp(p->data.name, name) != 0)
            p = p->next;
    }
    return p;
}

void DeleteNode(LinkList head)          //病患信息链表上节点的删除
{
    char jx;
    LNode *p,*q;
    p = ListFind(head);                 //调用查找函数
    if(p==NULL)
    {
        printf("没有查到要删除的病患");
        return;
    }
    printf("真的要删除该节点吗?(y/n):");
    scanf("%c",&jx);       //注意在%c 前加上一个空格,这样可以处理掉输入缓冲区
    fflush(stdin);         //这个函数也可以清除输入缓冲区
```

```
        if(jx == 'y' || jx == 'Y')
        {
            q = head;
            while(q != NULL && q->next != p)
                q = q->next;
            q->next=p->next;        //删除节点
            free(p);                //释放被删除的节点空间
            p = NULL;
            printf("病患已被删除\n");
        }
    }

    void PrintList(LinkList head)    //病患信息链表上输出函数
    {
        LNode *p;
        p=head->next;                //链表带头节点,使 p 指向链表开始节点
        printf("编号\t 姓名\t 性别\t 联系电话\t 地址\n");
        printf("-----------------------------------------\n");
        while(p != NULL)
        {
            printf("%s\t%s\t%s\t%s\t%s\n", p->data.num, p->data.name,
                p->data.sex, p->data.phone, p->data.addr);
            printf("-----------------------------------------\n");
            p = p->next;             //后移一个节点
        }
    }
```


独立实践

将新病人插入到链表的指定位置。

2.3.3 循环链表

在单链表中，将终端节点的指针域 NULL 改为指向表头节点或开始节点，就得到了单链表形式的循环链表，简称为单循环链表。在单循环链表中，表中所有节点被链接在一个环上。为了使空表和非空表的处理一致，循环链表中也可设置一个头节点。这样，空循环链表仅由一个自成循环的头节点表示。带头节点的单循环链表如图 2.16 所示。

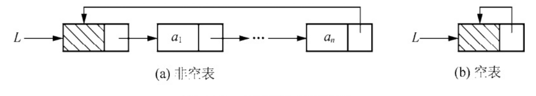


图 2.16 单循环链表示意图



在单循环链表上的操作基本上与非循环链表相同，只是将原来判断指针是否为 NULL 变为是否为头指针而已，没有其他变化。

例 2.2 求单循环链表的长度。

过程如图 2.17 所示，算法如下。

```
int Length(LinkList L)
{  int n;
   LinkList  p;
   p=L; n=0;
   while(p->next!=L)
   { p=p->next;
     ++n;
   }
  return(n);
}
```

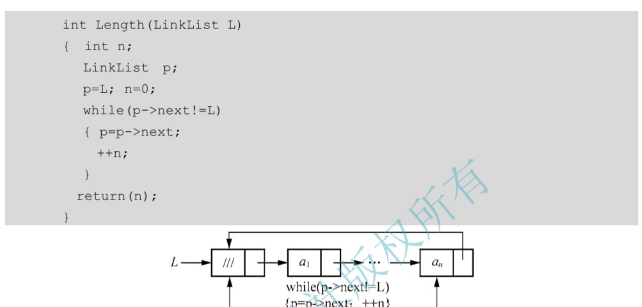


图 2.17　求单链表的长度示意图

在上述算法中，通过指针 p 从头节点扫描到终端节点，用 n 来计算节点个数，当指针 p 指向终端节点时表示统计结束，条件是 p->next 等于 L(若是单链表，则条件是 p->next 等于 NULL)。该算法的平均时间复杂度为 $O(n)$。

在用头指针表示的单循环链表中，找开始节点 a_1 的时间是 $O(1)$，然而要找到终端节点 a_n，则需从头指针开始遍历整个链表，其时间是 $O(n)$。在很多实际问题中，表的操作常常是在表的首尾位置上进行的，此时头指针表示的单循环链表就显得不够方便。如果改用尾指针 rear 来表示单循环链表(见图 2.18)，则查找开始节点 a_1 和终端节点 a_n 都很方便，它们的存储位置分别是 rear->next->next 和 rear，显然，查找时间都是 $O(1)$。因此，实际中多采用尾指针表示单循环链表。

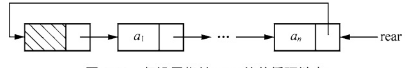


图 2.18　仅设尾指针 rear 的单循环链表

例 2.3 在链表上实现将两个线性表(a_1，a_2，…，a_n)和(b_1，b_2，…，b_m)链接成一个线性表(a_1，a_2，…，a_n，b_1，b_2，…，b_m)的运算。

若在带头指针表示的单循环链表上做这种链接操作，则需要遍历第一个链表，找到节点 a_n，然后将节点 b_1 链接到 a_n 的后面，其执行时间是 $O(n)$。若在尾指针表示的单循环链表上实现，则只需修改指针，无需遍历，其执行时间是 $O(1)$。指针修改过程如图 2.19 所示。相应的算法如下。

```
LinkList Connect(LinkList ra, LinkList rb)
{  LinkList *p;
   p=ra->next;                  //保存表 ra 的头节点地址
ra->next=rb->next->next;        //链表 rb 的开始节点链接到链表 ra 的终端节点之后
 free(rb->next);                //释放链表 rb 的头节点
 rb->next=p;                    //返回新循环链表的尾指针
}
```

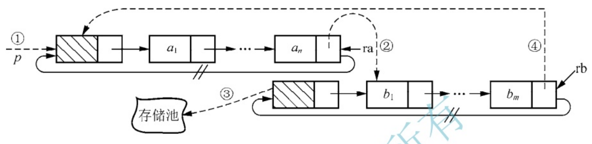


图 2.19　两个单循环链表的链接操作示意图

2.3.4　双链表

前面讨论的线性链表的节点中只有一个指向其后继节点的指针域 next，因此找到其后继非常方便，若已知某节点的指针为 p，其后继节点的指针则为 p->next，时间复杂度是 $O(1)$，而要找到其前趋节点，就只能从该链表的头指针开始，从前向后遍历，时间复杂度是 $O(n)$。如果希望找前趋的时间复杂度也是 $O(1)$，则可以用空间换取时间，即给每个节点再增加一个指向前趋的指针域，节点结构如图 2.20 所示。

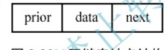


图 2.20　双链表结点结构

其中，每个节点除了数据域 data 外，包含了两个指针域：一个是 prior 指针，指向它的前趋节点；一个是 next 指针，指向该节点的后继节点。该节点的类型定义如下。

```
typedef struct DLNode
{ ElemType data;
  struct DLNode *prior,*next;
}DLNode,*DLinkList;
```

用这种两个指针域节点组成的链表称为双向链表，简称双链表。与单链表类似，双链表可以是非循环的，也可以是循环的。本书默认的是带头节点的双向循环链表，如图 2.21 所示。

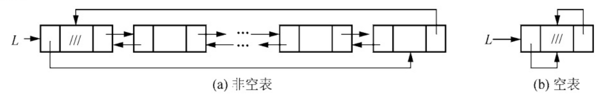


图 2.21　带头节点的双链表

若定义 p 是 DLinkList 类型的指针变量，设 p 指向双链表中的某一节点，如图 2.22 所示，则 p->next 表示 p 所指向节点的直接后继节点地址，而 p->prior 表示 p 直接前趋节点的地址。同时，有如下等价式。

p->prior->next ⇔ p ⇔ p->next->prior

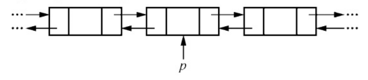


图 2.22　p 指向双链表的某节点

在双链表上的操作实现，求表长、元素定位、查找元素等操作仅涉及一个方向的指针，类似于单链表，这里请读者自行完成。在双链表中完成插入和删除操作则需要修改两个方向的指针。下面分别对其进行讨论。

1. 双链表中插入一个节点

在双链表中插入节点有多种方式，与单链表不同的是，在插入节点时无须知道插入位置的前趋节点，可以直接找到相应的节点，在其前面插入。设 p 指向链表中第 i 个节点，s 指向待插入的节点，将 s 节点插入到 p 节点的前面，插入过程如图 2.23 所示，语句描述如下。

① s->prior=p->prior;

② s->next=p;

③ p->prior->next=s;

④ p->prior=s;

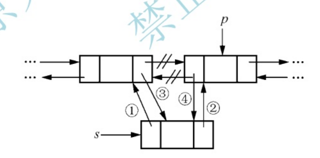


图 2.23　在 p 结点前插入 s 结点

需要强调的是，上述操作中的指针修改顺序并不是唯一的，但也不是任意的，操作过程中必须确保链表节点不丢失。

在双链表 L 中的第 i 个节点前插入一个元素 e，插入算法如下。

```
int ListInsert_Dul(DLinkList L, int i, ElemType e)
{ DLinkList p;
  int j;
  p=L; j=0;
  while(p && j<i)  //寻找第 i 个节点,并令 p 指向它
```

```
    { p=p->next; ++j; }
  if( !p || j>i) return FALSE;
  s=(DLinkList)malloc(sizeof(DLNode));
  s->data=e;
  s->prior=p->prior;  s->next=p;
  p->prior->next=s;   p->prior=s;
  return TRUE;
}
```

该算法时间主要用在查找第 i 个节点上，时间复杂度为 $O(n)$。

2. 双链表中删除一个节点

在双链表中删除节点也有多种方法，与单链表不同的是，也无需知道被删除节点的前趋节点，直接找到被删除的节点，然后执行相关的指针修改操纵即可。以删除第 i 个节点为例，设 p 指向双向链表中第 i 个节点，删除 p 节点的操作过程如图 2.24 所示，语句描述如下：

① p->prior->next=p->next;

② p->next->prior = p->prior ;

③ free(p);

删除双链表 L 中的第 i 个节点的算法如下。

```
int ListDelete_DuL(DLinkList L, int i)
{ DLinkList p;
  int j;
   p=L; j=0;
   while(p && j<i)
    { p=p->next; ++j;}
   if( !p || j>i)  return FALSE;
   p->prior->next=p->next;
   p->next->prior=p->prior;
   free(p);
   return TRUE;
}
```

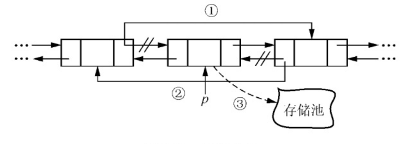
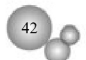


图 2.24　删除结点 p

同插入算法相似，删除算法的主要时间消耗在查找第 i 个节点上，时间复杂度为 $O(n)$。



2.4 顺序表和链表的比较

前面两节介绍了线性表的两种存储结构：顺序表和链表，它们各有优缺点。在实际应用中选用哪一种存储结构主要取决于具体问题的要求和性质。通常有以下两方面的考虑。

1. 从空间上

顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模。若线性表的长度 n 变化较大，则存储规模难以预先确定。估计过大将造成空间浪费，估计过小又将使空间溢出几率增大。而动态链表的存储空间是动态分配的，只要内存空间尚有空闲，就不会产生溢出。因此，当线性表的长度变化较大，难以估计其存储规模时，采用动态链表作为存储结构为宜。

2. 从时间上

顺序表是由向量实现的，它是一种随机存取结构，对表中任一节点都可在 $O(1)$时间被直接存取，而链表中的节点，需从头指针起顺着链表扫描才能取得。因此，当线性表的操作主要是进行查找，很少做插入和删除操作时，采用顺序表做存储结构为宜。

在链表中的任何位置上进行插入和删除，都需要修改指针。而在顺序表中进行插入和删除，平均要移动表中近一半的节点，尤其是当每个节点的信息量较大时，移动节点的时间开销就相当可观。因此，对于频繁进行插入和删除的线性表，宜采用链表做存储结构。若表的插入和删除主要发生在表的首尾两端，则采用尾指针表示的单循环链表为宜。

小 结

线性表是一种最基本、最常用的数据结构。本章介绍了线性表的定义、运算和各种存储结构的描述方法，重点讨论了线性表的两种存储结构——顺序表和链表，以及在这两种存储结构上实现的基本运算。

顺序表是用数组实现的，链表是用指针实现的。在实际应用中，对线性表采用哪种存储结构，要视实际问题的要求而定，主要考虑求解算法的时间复杂度和空间复杂度，因此，建议读者熟练掌握在顺序表和链表上实现的各种基本运算及其时间、空间特性。

习 题

一、填空题

1．向一个长度为 n 的向量的第 i 个元素($1\leqslant i\leqslant n+1$)之前插入一个元素时，需向后移动________个元素。

2．向一个长度为 n 的向量中删除第 i 个元素($1\leqslant i\leqslant n$)时，需向前移动________个元素。

3．在顺序表中访问任意一个节点的时间复杂度均为________，因此，顺序表也称为________的数据结构。

4．顺序表中逻辑上相邻的元素的物理位置________相邻。单链表中逻辑上相邻的元素的物理位置________相邻。

5．在 n 个节点的单链表中要删除已知节点*p，需找到它的________，其时间复杂度为________。

二、判断题

1．链表的物理存储结构具有同链表一样的顺序。　(　　)

2．链表的删除算法很简单，因为当删除链中的某个节点后，计算机会自动将后续各个单元向前移动。　(　　)

3．顺序表结构适宜进行顺序存取，而链表适宜进行随机存取。　(　　)

4．线性表在物理存储空间中也一定是连续的。　(　　)

5．线性表在顺序存储时，逻辑上相邻的元素未必在存储的物理位置次序上相邻。(　　)

6．线性表的逻辑顺序与存储顺序总是一致的。　(　　)

三、选择题

1．一个向量第一个元素的存储地址是 100，每个元素的长度为 2，则第 5 个元素的地址是(　　)。

A．110　　B．108　　C．100　　D．120

2．在 n 个节点的顺序表中，算法的时间复杂度是 $O(1)$的操作是(　　)。

A．访问第 i 个节点($1 \leqslant i \leqslant n$)和求第 i 个节点的直接前趋($2 \leqslant i \leqslant n$)

B．在第 i 个节点后插入一个新节点($1 \leqslant i \leqslant n$)

C．删除第 i 个节点($1 \leqslant i \leqslant n$)

D．将 n 个节点从小到大排序

3．链接存储的存储结构所占存储空间(　　)。

A．分两部分，一部分存放节点值，另一部分存放表示节点间关系的指针

B．只有一部分，存放节点值

C．只有一部分，存储表示节点间关系的指针

D．分两部分，一部分存放节点值，另一部分存放节点所占单元数

4．线性表若采用链式存储结构时，要求内存中可用存储单元的地址(　　)。

A．必须是连续的　　B．部分地址必须是连续的

C．一定是不连续的　　D．连续或不连续都可以

5．线性表 L 在(　　)情况下适用于使用链式结构实现。

A．需经常修改 L 中的节点值　　B．需不断对 L 进行删除插入

C．L 中含有大量的节点　　D．L 中节点结构复杂

6．线性表是(　　)。

A．一个有限序列，可以为空　　B．一个有限序列，不能为空

C．一个无限序列，可以为空　　D．一个无序序列，不能为空

7．用链表表示线性表的优点是(　　)。

A．便于随机存取

B．花费的存储空间较顺序存储少

C．便于插入和删除

D．数据元素的物理顺序与逻辑顺序相同

8．对顺序存储的线性表，设其长度为 n，在任何位置上插入或删除操作都是等概率的。插入一个元素时平均要移动表中的(　　)个元素。

A．$n/2$　　B．$(n+1)/2$　　C．$(n-1)/2$　　D．n

四、简答题

1．试比较顺序存储结构和链式存储结构的优缺点。在什么情况下用顺序表比链表好？

2．描述以下三个概念的区别：头指针、头节点、开始节点。在单链表中设置头节点的作用是什么？

五、编程题

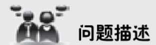
1．设顺序表 L 中的数据元素递增有序。试编写一算法，将 x 插入到顺序表的适当位置，以保持该表的有序性。

2．分别用顺序表和单链表存储结构实现线性表的就地逆置。线性表的就地逆置就是在原表的存储空间内将线性表(a_1，a_2，a_3，…，a_n)逆置为(a_n，a_{n-1}，…，a_2，a_1)。

3．设指针 s 指向单循环链表的一个节点，请设计算法，删除该节点的前趋节点。



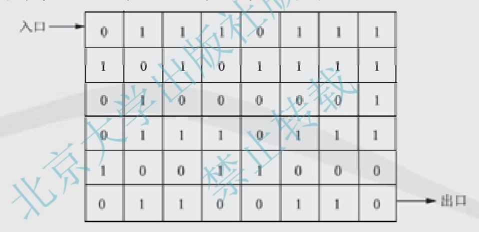
栈 与 队 列



问题描述

迷宫求解问题

心理学家把一只老鼠从迷宫(如图 3.1 所示)的入口赶入，迷宫中设置很多墙壁(设 0 为通道，1 为墙壁)，对前进方向形成了多处障碍。并在迷宫的唯一出口放置了一块奶酪，吸引老鼠在迷宫中寻找通路以到达出口。请编程求出迷宫的通路。



0	1	1	1	0	1	1	1
1	0	1	0	1	1	1	1
0	1	0	0	0	0	0	1
0	1	1	1	0	1	1	1
1	0	0	1	1	0	0	0
0	1	1	0	0	1	1	0

图 3.1　迷宫

3.1　栈

3.1.1　栈的定义

堆栈(Stack)也简称为栈，是限定仅在表的一端进行插入和删除操作的线性表。通常将进行插入和删除的一端(表尾)叫做栈顶(top)，不允许插入和删除的另一端(表头)叫做栈底(bottom)，不含任何元素的栈叫做空栈。

插入数据元素的操作叫做进栈，也称压栈、入栈。删除数据元素的操作叫做出栈，也称为退栈。由于堆栈元素的插入和删除只是在栈顶进行的，总是后进去的元素先出来，所以堆栈又称为后进先出线性表或 LIFO(Last In First Out)表。堆栈在日常生活中也经常见到，如将子弹装入弹夹式的手枪，弹夹的一端是封闭的，子弹的放入和取出都是从弹夹的一端进行的，它就是一个堆栈。

理解堆栈的定义时需要注意：它是一个线性表，也就是说，栈元素具有线性关系，即前趋后继关系。只不过它是一种特殊的线性表；定义中所说的在线性表的表尾进行插入和删除操作，这里的表尾是指栈顶，而不是栈底。我们可以用图 3.2 来形象地说明。

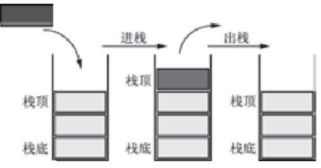


图 3.2　进出栈示意图

3.1.2　栈的基本操作

对于栈的基本运算，常见的有以下几种。

```
InitStack(&S)            //初始化栈:构造一个空栈 S
ClearStack(&S)           //置空栈:把 S 置为空栈
StackLength(S)           //求栈的长度:返回栈 S 中的元素个数
StackEmpty(S)            //判断栈是否为空:若栈 S 为空,则返回真;否则返回假
Push(&S,e)               //进栈:将插入元素 e 为新的栈顶元素
Pop(&S,&e)               //出栈:若栈不空,则删除 S 的栈顶元素,用 e 返回其值
GetTop(S,&e)             //取栈顶元素:返回当前的栈顶元素,并将其赋值给 e
DispStack(S)             //显示元素:从栈底到栈顶依次显示栈中每个元素
```


栈有两种存储方式：顺序存储(顺序栈)和链式存储(链栈)。在实际应用中，以顺序存储的栈为主，故下面主要介绍顺序栈。

3.1.3　栈的顺序存储和实现

1．栈的顺序存储

顺序栈类似于顺序表，用一维数组来存放栈中元素，栈底一般固定设在下标为 0 的一端，用一个变量 top 指示当前栈顶元素所在单元的位置。

通常情况下，把空栈的判定条件定位为 top=-1，当栈中有一个元素时，top=0；而栈满时，top=MAXSIZE-1，栈的长度等于 top+1。示意图如图 3.3 所示。其进出栈的实现见算法 3.1 和算法 3.2。

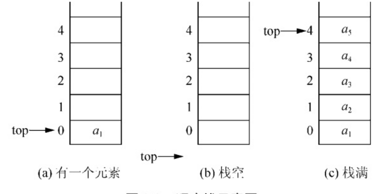


图 3.3　顺序栈示意图

顺序栈的结构定义如下。

```
#define MAXSIZE 100        /* 设顺序栈的最大长度为100,可根据实际情况做修改 */
typedef int DataType;
typedef struct{
DataType data[MAXSIZE];
int top;                   /* 用于栈顶指针 */
}SqStack;
```

2. 栈的运算实现

1) 初始化栈

建立一个新的空栈 *S*，实际上将栈顶指针指向-1 即可。

算法 3.1 栈的顺序存储的初始化操作。

```
int InitStack(SqStack *S)
{
    S->top = -1;
    return TRUE;
}
```

2) 清空栈

清空栈中的元素，此时栈顶指针指向-1。

算法 3.2 栈的顺序存储的置空操作。

```
int ClearStack(SqStack *S)
{
    S->top = -1;
    return TRUE;
}
```


3) 判断栈是否为空

判断栈是否为空时，若栈 *S* 为空栈，则返回 TRUE，否则返回 FALSE。

算法 3.3 栈的顺序存储的判空操作。

```
int StackEmpty(SqStack S)
{
    if (S.top == -1)
        return TRUE;
    else
        return FALSE;
}
```

4) 求栈的长度。

返回 *S* 的元素个数，即栈的长度。

算法 3.4 栈的顺序存储求栈的长度操作。

```
int StackLength(SqStack S)
```

```
    {
        return S.top+1;
    }
```

5) 返回栈顶元素

若栈不空，则用 *e* 返回 *S* 的栈顶元素，并返回 TRUE；否则返回 FALSE。

算法 3.5 栈的顺序存储返回栈顶元素操作。

```
    int GetTop(SqStack S, DataType *e)
    {
        if (S.top==-1)
            {   printf("栈空!");
                return FALSE;
            }
        else
            *e=S.data[S.top];
        return TRUE;
    }
```

6) 显示元素

从栈底到栈顶依次显示栈中每个元素。

算法 3.6 栈的顺序存储显示栈元素操作。

```
    int DispStack(SqStack S)
    {
        int i=0;
        if(s.top==-1)
        {   prinft ("栈空!");
            return FALSE;
        }
        while(i<=S.top)
        {
            printf("%d", S.data[i++]);
        }
        printf("\n");
        return TRUE;
    }
```


7) 入栈操作

对于栈的插入，即入栈操作，其算法执行步骤描述如下。

(1) 判断栈是否已满。

(2) 如果栈没满，则让栈顶指针上移。

(3) 数据元素入栈。

实际上，就是做了如图 3.2 和图 3.3 所示的处理，对于入栈操作 push，插入元素 *e* 为新的栈顶元素，其算法实现代码如下.

算法 3.7 栈的顺序存储的入栈操作。

```
    int Push(SqStack *S, int e)
    {
    if(S->top == MAXSIZE -1)     //栈满
    {
```

```
        {   printf ("栈空!");
            return FALSE;
        }
      S->top++;                  //栈顶指针增加 1
      S->data[S->top]=e;         //将新插入的元素赋值给栈顶空间
      return TRUE;
    }
```

8) 出栈操作

出栈操作算法的执行步骤描述如下。

(1) 判断栈是否为空。

(2) 如果栈不为空，则取出栈顶元素值。

(3) 栈顶指针下移。

出栈操作 pop，若栈不空，则删除 *S* 的栈顶元素，用 *e* 返回其值，并返回 TRUE；否则返回 ERROR，其算法实现源程序代码如下。

算法 3.8 栈的顺序存储的出栈操作.

```
    int Pop(SqStack *S, int *e)
    {
        if(S->top==-1)
        {   printf ("栈空!");
            return FALSE;
        }
        *e=S->data[S->top];          // 将要删除的栈顶元素赋值给 e
        S->top--;                    // 栈顶指针减 1
        return TRUE;
    }
```

比较顺序栈入栈和出栈的过程，可以看出，入栈先移动栈顶指针而后插入元素，出栈是先取出原栈顶元素后才移动栈顶指针。

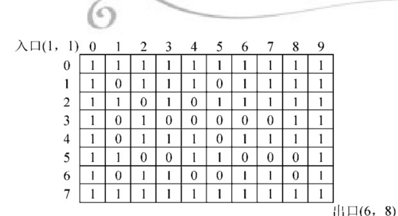
3.1.4 用栈实现的迷宫问题

迷宫问题的求解可以采用回溯法，即一种不断试探且及时纠正错误的搜索方法。其思想如下：从迷宫入口出发，按某一方向向前探索，若能走通(未走过的)，即某处可以到达，则到达新点，否则试探下一方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向继续试探，直到所有可能的通路都探索到，或找到一条通路，或无路可走又返回到入口点。

在求解过程中，为了保证在到达某一点后不能向前继续行走(无路)时，能正确返回前一点以便继续从下一个方向向前试探，则需要用一个栈保存所能够到达的每一点的下标及到达该点的方向。因此，需要解决以下四个问题。

1) 表示迷宫的数据结构

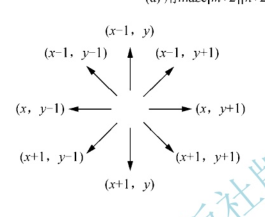
设迷宫为 *m* 行 *n* 列，利用 maze[*m*][*n*]来表示一个迷宫，maze[*i*][*j*]=0 或 1，其中，0 表示通路，1 表示不通。当从某点向前试探时，中间点有八个方向(见图 3.4(b))可以试探，而四个角点有三个方向，其他边缘点有五个方向。为使问题简单化，我们用 maze[*m*+2][*n*+2]来表示迷宫，即在迷宫四周设立“哨兵”，则迷宫四周的值全部为 1。这样将使问题简单化，每个点的试探方向全部为 8，不用再判断当前点的试探方向有几个，同时“哨兵”与迷宫周围是墙壁这一实际问题相一致。



入口(1，1)	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	0	1	1	1	1
2	1	1	0	1	0	1	1	1	1	1
3	1	0	1	0	0	0	0	0	1	1
4	1	0	1	1	1	0	1	1	1	1
5	1	1	0	0	1	1	0	0	0	1
6	1	0	1	1	0	0	1	1	0	1
7	1	1	1	1	1	1	1	1	1	1

出口(6，8)

(a) 用maze[*m*+2][*n*+2]表示的迷宫



(b) 与点(*x*，*y*)相邻的八个点及坐标

	x	*y*
0	0	1
1	1	1
2	1	0
3	1	−1
4	0	−1
5	−1	−1
6	−1	0
7	−1	1

(c) 增量数组move

图 3.4　迷宫示意图

图 3.4(a)表示的是一个 6×8 的迷宫。入口坐标为(1，1)，出口坐标为(6，8)，则该迷宫的定义如下。

```
#define  m  6              //迷宫的实际行数
#define  n  8              //迷宫的实际列数
int maze[m+2][n+2];        //二维数组存放迷宫
```

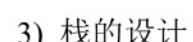
2) 试探方向

在上述表示迷宫的情况下，每个点有八个方向去试探，与其相邻的八个点的坐标都可根据与该点的相对方位而得到。试探顺序规定为：从当前位置向前试探的方向为从正东沿顺时针方向进行。为了简化问题，将从正东开始沿顺时针进行的这八个方向的坐标增量顺序存放在一个结构数组 move[8]中，在 move 数组中，每个元素由两个域组成，*x* 是横坐标增量，*y* 是纵坐标增量。move 数组如图 3.3(c)所示，则定义如下。

```
typedef  struct
{
    int x,y;
}Item;
Item move[8] ;
```

这样可以很方便地求出从某点(*x*，*y*)按某一方向 $v(0\leqslant v\leqslant 7)$到达的新点(*i*，*j*)的坐标：

```
    i=x+move[v].x;  j=y+move[v].y;
```



3) 栈的设计

当到达了某点而无路可走时需返回前一点，再从前一点开始沿下一个方向继续试探。因此，压入栈中的不仅要有顺序到达的各点的坐标，还要有从前一点到达本点的方向。

栈中每一组数据是所到达的每点的坐标及从前一点到达本点的方向。对于图 3.3 所示的迷宫，走的路线为：(1，1)$_1$→(2，2)$_1$→(3，3)$_1$→(3，4)$_0$→(3，5)$_0$→(3，6)$_0$→(3，7)$_0$(下脚标表示方向)，当从点(3，6)沿方向 0 到达点(3，7)之后，无路可走，则应回溯，即退回到点(3，6)。对应的操作是出栈，沿下一个方向，即方向 1 继续试探，方向 1、2 试探失败，在方向 3 上试探成功，即到达了(4，5)点，因此将(4，5，3)压入栈中，再继续向前搜索。最后将(6，8，2)压入栈中，因为已到达出口，找到了一条出路，所以算法结束。

因此，栈中元素是一个由行、列、方向组成的三元组，设计如下。

```
typedef  struct
{
    int x,y,d;   //横纵坐标及方向
}SElemType;
```

顺序栈类型定义为如下。

```
typedef struct
{    SElemType   stack[STACK_INIT_SIZE];
     int  top;
}SqStack;     // 定义顺序栈类型
```

4) 如何避免发生死循环

避免发生死循环有两种方法：一种方法是另外设置一个标志数组 mark[*m*][*n*]，将它的所有元素都初始化为 0，一旦到达了某一点(*i*，*j*)，就使 mark[*i*][*j*]置 1，下次再试探这个位置时就不能继续往前了；另一种方法是，当到达某点(*i*，*j*)后使 maze[*i*][*j*]置-1，以便区别不同的点，同样也能起到防止走重复点的目的。本实例采用后一种方法，算法结束前可再恢复原迷宫。

迷宫求解算法的思想如下。

(1) 栈初始化。

(2) 将入口点坐标及到达该点的方向(设为-1)入栈，初始搜索方向 dd 为第一个搜索方向 0。

(3) 若栈不空，重复如下操作。

① 取栈顶元素。

② 从栈顶元素对应的顶点开始，依次试探各个没有考查过的方向，直至找到一个可行顶点或无可行顶点为止。若有，则将其入栈，再初始搜索方向 dd 为第一个搜索方向 0，若此时已到达出口，说明已找到一条路径，输出该路径，算法结束；若没有可行顶点，出栈，退回到上一步，搜索方向 dd 为上一次搜索的下一个方向。

(4) 若最后栈为空，则表明没有出路，搜索失败；栈非空，表明搜索成功，搜索路径上的顶点放在栈 *S* 中。最后，栈中保存的就是一条迷宫的通路。

程序实现如下。

```
#include <stdio.h>
#include <stdlib.h>
#define m 6                          // 迷宫的实际行
#define  n  8                        // 迷宫的实际列
```

```
#define STACK_INIT_SIZE 100       //存储空间初始分配量
#define STACKINCREMENT 10         //存储空间分配增量
typedef  struct
{ int x,y,d;                      // 横纵坐标及方向
}SElemType;

typedef struct
{   SElemType   stack[STACK_INIT_SIZE];
int  top;
}SqStack;                         // 定义顺序栈类型

typedef  struct
{ int  x,y;
}Item;
SElemType  XX={-1,-1,-1};
int maze[m+2][n+2]=
{ /*      0,1,2,3,4,5,6,7,8,9   */
/*0*/   {1,1,1,1,1,1,1,1,1,1},
/*1*/   {1,0,1,1,1,0,1,1,1,1},
/*2*/   {1,1,0,1,0,1,1,1,1,1},
/*3*/   {1,0,1,0,0,0,0,0,1,1},
/*4*/   {1,0,1,1,1,0,1,1,1,1},
/*5*/   {1,1,0,0,1,1,0,0,0,1},
/*6*/   {1,0,1,1,0,0,1,1,0,1},
/*7*/   {1,1,1,1,1,1,1,1,1,1}};
Item move[8]={{0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1},{-1,0},{-1,1}};
//八个移动方向

int Empty_SqStack(SqStack *s);
void Push_SqStack(SqStack *s,SElemType x);
SElemType Gettop_SqStack(SqStack *s);
SElemType Pop_SqStack(SqStack *s);
int* OnePath(int maze[m+2][n+2],Item move[8],SqStack *s);

void main( )
{
    SElemType temp;
    SqStack *s;
    s =(SqStack *)malloc(sizeof(SqStack));
    if(!s)
        {exit(0);}
    else
        {s->top = -1;}
    OnePath(maze,move,s);              // 探索路径,函数内容见主算法
    if(s->top==-1)                     // 栈为空,迷宫无出路
        printf ("此迷宫无路径\n");
    else                               // 栈非空,给出迷宫路径
    {
```

```
        printf ("迷宫路径为: \n") ;
        while(!Empty_SqStack(s))     // 栈非空,将栈中的元素出栈并输出
        {
            temp=Pop_SqStack(s);
            if(s->top==-1)
                printf("(%d,%d)",temp.x,temp.y);
            else
                printf("(%d,%d)-",temp.x,temp.y);
        }
        printf("\n 走出迷宫啦! \n");
        while(getchar()!='0')
        {
            getchar();
            printf("请输入 0,退出程序! \n");
        }
    }
}

int Empty SqStack(SqStack *s)
{ // 判栈空。要求:空栈栈顶指针为 0 时,返回 1,否则返回 0
    if(s->top==-1)
        return 1;
    else
        return 0;
}

void Push_SqStack(SqStack *s,SElemType x)
{ //将元素 x 压入栈 s 中。要求:栈满时,提示栈满信息,完成入栈操作则返回
    if (s->top==STACK_INIT_SIZE-1)
    {
        printf("\n 栈满!");
    }
    else
    {
        s->top++;
        s->stack[s->top]=x;
    }
}

SElemType Gettop_SqStack(SqStack *s)
{/*  取栈顶元素。要求:栈空时不能出栈,返回特殊值 XX;否则返回栈顶元素*/
    if(s->top==-1)
        return XX;
    else
        return s->stack[s->top];
}

SElemType Pop_SqStack(SqStack *s)
```

```
 {/* 出栈,成功时返回顶元素;不成功时结束程序*/
   if(Empty_SqStack(s))
     {
printf("\n 栈为空! ");
          exit(0);
     }
     else
     {
        s->top--;
        return s->stack[s->top+1];
     }
}

int* OnePath(int maze[m+2][n+2],Item move[8],SqStack *s)
{/*探索一条迷宫路径,成功时返回 1,否则返回 0。探索成功时,将搜索路径上的顶点放入栈 S 中*/
     int x,y,d,i,j,dd;
     SElemType temp;
     temp.x=1;  temp.y=1;temp.d=-1;// 入口点坐标及到达该点的方向(设为-1)入栈
     Push_SqStack(s,temp);                      //temp 入栈
     maze[temp.x][temp.y]=-1; dd=0;             //第一个搜索方向
     while(!Empty_SqStack(s))
     {
          temp=Gettop_SqStack(s);               //取栈顶元素
          x=temp.x;  y=temp.y;
          d=dd;
          while(d<8)
          {
               i=x+move[d].x; j=y+move[d].y;
               if(maze[i][j]==0)                //找到下一步的一个可行顶点
               {
               temp.x=i; temp.y=j; temp.d=d;    // 下一步的一个顶点 temp
                    Push_SqStack(s,temp);       //顶点入栈
                    dd=0;  // 搜索方向初始化为第一个方向
                    maze[i][j]= -1;             // 给已走过的路径做标记
                     if(i==m && j==n) return 0; // 找到迷宫出口,结束
                     break;  // 找到下一步的一个顶点之后,就暂时不再沿其他方向搜索
               }
                else  d=d+1;                    // 探索下一个方向
          }
          if(d==8)                              // 无路可走
          {
           temp= Pop_SqStack(s);
               dd=temp.d+1;  // 再沿上一步的下一个方向搜索
          }
     }
     return 0;
}
```



独立实践

在本例中，采用栈结构用回溯法探索迷宫路径，过程比较清晰，但只能找到所有可能的路径中的一条。请尝试修改本例的算法，以便求出所有可行的路径。

3.2 队　　列



问题描述

银行排队叫号问题

在以银行营业大厅为代表的窗口业务中，大量客户的拥挤排队已成为了这些企事业单位改善服务品质、提升企业形象的主要障碍。排队叫号系统的使用将成为改变这种情况的有力手段。排队系统完全模拟了人群排队的全过程，通过取票进队、排队等待、叫号服务等功能，很好地解决了客户在服务机构办理业务时所遇到的各种排队、拥挤和混乱现象。

3.2.1　队列的概念

队列(Queue)又是一种运算受限制的线性表，它只允许在表的一端进行插入，而在另一端进行删除。允许插入元素的一端称为队尾(Rear)；允许删除元素的一端称为队头(Front)；不含元素的空表称为空队列。向队列添加元素称为入队，从队列中删除元素称为出队。由于新入队的元素只能添加到队尾，出队的元素只能删除队头的元素，所以队列的特点是先进入队列的元素先出队，故队列也称为先进先出表或 FIFO(First In First Out)表。

在日常生活中有很多这样的例子，如移动、联通、电信等客服电话，客服人员与客户相比总是少数，在所有的客服人员都占线的情况下，客户会被要求等待，直到有某个客服人员空闲，才能让最先等待的客户接通电话。这里就是将所有当前拨打客服电话的客户进行了排队处理。假设队列是 $q=(a_1, a_2, \cdots, a_n)$，那么 a_1 就是队头元素，而 a_n 就是队尾元素。这样就可以在删除时，总是从 a_1 开始，而插入时，列在最后，如图 3.5 所示。

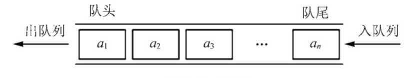


图 3.5　队列

3.2.2　队列的基本操作

队列的基本操作主要包括如下几部分。

```
InitQueue (&Q)                    //初始化空队列 Q
ClearQueue (&Q)                   //释放栈 Q 占用的存储空间
QueueEmpty (Q)                    //判断队列是否为空
QueueLength(Q)                    //返回队列的长度
GetHead(Q,&e)                     //用 e 返回 Q 的队头元素
```

```
    EnQueue(&Q,e)                    //将元素 e 入队作为队尾元素
    DeQueue(&Q,&e)                   //从队列 Q 中出队一个元素,用 e 返回
    QueueTraverse(Q)                 //从队头到队尾依次对队列 Q 中每个元素输出
```

队列也有两种存储方式：顺序存储(顺序队列)和链式存储(链队列)。

3.2.3 队列的顺序存储、实现和应用

1. 顺序队列

顺序存储的队列称为顺序队列。顺序队列类似于顺序表，用一维数组来存放队列元素，实际上就是运算受限的顺序表。但由于队头和队尾都是活动的，因此，设有两个指针：队头指针(front)和队尾指针(rear)。为方便起见，我们规定队头指针 front 总是指向当前队头元素的位置，队尾指针 rear 指向当前队尾元素的后一位置，如图 3.6 所示。

当队列中无元素时，则称其为空队列，并规定此时队头指针和队尾指针均为 0，即 front = rear = 0。每当插入新的队列尾元素时，“尾指针增 1”；每当删除队头元素时，“头指针增 1”。

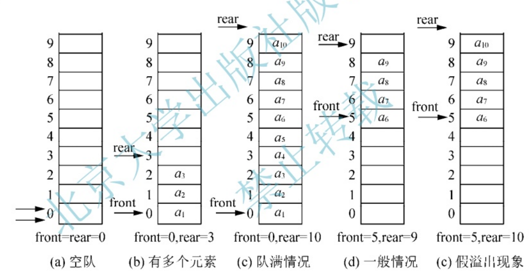


图 3.6 队列操作示意图

顺序队列的存储结构定义如下。

```
#define MAXSIZE 100
typedef int DataType;
typedef struct
{   DataType data[MAXSIZE];
    int front;          //头指针
    int rear;           //尾指针,若队列不空,则指向队列尾元素的下一个位置
}SqQueue;
```

图 3.7 中说明了在顺序队列中进行出队和入队运算时队列中的元素及头尾指针的变化情况。一开始，队列的头、尾指针都指向向量空间下标为 0 的位置，若不考虑溢出，则入队运算可描述为：

```
sq->data[sq->rear]=x;     /*  x 入队  */
sq->rear++;               /*  尾指针加 1  */
```

出队运算可描述为：

```
sq->front++;              /*  头指针加 1  */
```

显然，当前队列中的元素个数(即队列的长度)是(sq->rear)- (sq->front)。若sq->front=sq->rear，则队列长度为 0，即当前队列是空队列，如图 3.6(a)表示空队列。空队列时再做出队操作便会产生“下溢”。队满的条件是当前队列长度等于向量空间的大小，即：

```
(sq->rear)- (sq->front)=MAXSIZE
```

如图 3.6(c)所示，队满时再做入队操作会产生“上溢”。但是，如果当前尾指针等于向量的上界(即 sq->rear=MAXSIZE-1)，即使队列不满(即当前队列长度小于 MAXSIZE)，再作入队操作也会引起上溢。例如，若图 3.6(e)是当前队列的状态，即 MAXSIZE=10，sq->rear=10，sq->front=5，因为 sq->rear+1＞MAXSIZE，故此时不能作入队操作，但当前队列并不满，我们把这种现象称为“假上溢”。产生该现象的原因是，被删元素的空间在该元素删除以后就永远使用不到。为克服这一缺点，可以在每次出队时将整个队列中的元素向前移动一个位置，也可以在发生假上溢时将整个队列中的元素向前移动直至头指针为 0，但这两种方法都会引起大量元素的移动，所以在实际应用中很少采用。通常采用循环队列的方法来解决假溢出问题。

2. 循环队列

设想向量 sq->data[MAXSIZE]是一个首尾相接的圆环，即 sq->data[0]接在sq->data[MAXSIZE−1]之后，我们将这种意义下的向量称循环向量，并将循环向量中的队列称为循环队列，如图 3.7 所示，若当前尾指针等于向量的上界，则再做入队操作时，令尾指针等于向量的下界，这样就能利用到已被删除的元素空间，克服假上溢现象。因此入队操作时，在循环意义下的尾指针加 1 操作可描述为：

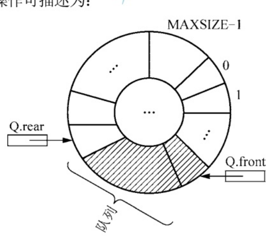


图 3.7 循环队列

```
if (sq->rear+1>=MAXSIZE )          sq->rear=0;
else sq->rear++;
```

如果利用“模运算”，上述循环意义下的尾指针加 1 操作，可以更简洁地描述为：

```
sq->rear=(sq->rear+1) % MAXSIZE
```

同样，出队操作时，在循环意义下的头指针加 1 操作，也可利用“模运算”来实现：

```
sq->front=(sq->front+1) % MAXSIZE
```

因为出队和入队分别要将头指针和尾指针在循环意义下加 1，所以某一元素出队后，若头指针已从后面追上尾指针，即 sq->front=sq->rear，则当前队列为空；若某一元素入队后，尾指针已从后面追上头指针，即 sq->rear=sq->front，则当前队列为满。因此，仅凭等式 sq->front=sq->rear 是无法区别循环队列是空还是满的。对此，有两种解决的办法：其一是引入一个标志变量以区别是空队还是满队；另一种更为简单的办法是：入队前，测试尾指针在循环意义下加 1 后是否等于头指针，若相等则认为是队满，即判别队满的条件是：

```
(sq->rear+1)%MAXSIZE ==sq->front
```

从而保证了 sq->rear==sq->front 是队空的判别条件。应当注意，这里规定的队满条件使得循环向量中，始终有一个元素的空间(即 sq->data[sq->rear])是空的，即有 MAXSIZE 个分量的循环向量只能表示长度不超过 MAXSIZE-1 的队列。这样做避免了由于判别另设的标志而造成时间上的损失。

在循环队列上实现的五种基本运算如下：

(1) 置空队

初始化一个空循环队列 sq，即将 sq 的 front 指针和 rear 指针设置为 0。

算法 3.9 循环队列的初始化

```
int InitQueue(SqQueue *sq)   /* 置队列 sq 为空队*/
{
    sq->front = 0;
    sq->rear = 0;
    return TRUE;
}
```

(2) 判队空

若循环队列 sq 为空队列，则返回 TRUE，否则返回 FALSE。

算法 3.10 循环队列的判空

```
int QueueEmpty(SqQueue sq)
{
    if(sq.front==sq.rear)      /* 队列空的标志 */
        return TRUE;
    else
        return FALSE;
}
```

(3) 循环队列的长度

返回队列 sq 的元素个数，也就是队列的当前长度。

算法 3.11 循环队列的长度

```
int QueueLength(SqQueue sq){
    return  (sq.rear-sq.front+MAXSIZE)%MAXSIZE;
}
```

(4) 返回循环队列的首元素

若队列不空，则用 e 返回 sq 的队头元素，并返回 TRUE，否则返回 ERROR。

算法 3.12 返回循环队列的首元素

```
int GetHead(SqQueue sq,DataType *e)
{
    if(sq.front==sq.rear)   /*判断队列是否空*/
    { printf ("空队列!\n");
      return FALSE;
    }
    *e=sq.data[sq.front];
    return TRUE;
}
```

(5) 循环队列的入队操作

将待添加的元素变量 e 插入到队列 sq 中，需注意，要将元素 e 先插入到队尾指针 rear 所指向的位置，再将队尾指针 rear 加 1。

算法 3.13 循环队列的入队操作

```
int EnQueue ( SqQueue *sq, DataType e)
{
    if (sq->front==(sq->rear+1)%MAXSIZE)     /* 判断队列是否已满*/
        printf ("溢出!\n");
    return FALSE
    {
        sq->data[sq->rear] = e;              /* 插入元素 e */
        sq->rear=(sq->rear+1) % MAXSIZE;
    }
     return TRUE;
}
```

(6) 循环队列的出队操作

当从队列删除元素时，队头指针 front 后移而队尾指针 rear 不动，做出队运算时，假设要求将出队的元素值赋给变量 e。若队列不空，出队操作是先把被删除的队头元素，用 e 返回其值，再将队头指针加 1，表明队头元素出队。

算法 3.14 循环队列的出队操作

```
int DeQueue ( SqQueue *sq, DataType *e)
{
    if (sq->front == sq->rear)               /*队列空的判断  */
    {   printf ("队列空!\n");
```

```
            return FALSE ;
        }
        *e= sq->data[sq->front];                    /*将队头元素赋值给 e  */
        sq->front=(sq-> front +1) % MAXSIZE;        /*front 指针向后移一位置*/
        return TRUE;
    }
```

(7) 循环队列的遍历

从队头到队尾依次输出队列 sq 中每个元素。

算法 3.15　循环队列的元素显示

```
    int QueueTraverse(SqQueue sq)
    {
        int i;
        if(sq.front==sq.rear)
        {
            printf("队列空!\n");
            return FALSE
        }
        i=sq.front;                         /*i 最初指向队头元素*/
        while(i!=sq.rear)
        {
            printf("%d ", sq.data[i]);
            i=(i+1)%MAXSIZE;                /*i 指向下一个元素*/
        }
        printf("\n");
        return TRUE;
    }
```


从这一小节中，我们发现单是顺序存储，若不是循环队列，算法的时间性能是不高的，但循环队列又面临着数组可能会溢出的问题，所以可以用链式存储结构来实现队列。

3.2.4　队列的链式存储、实现和应用

1. 队列的链式存储结构

队列的链式存储结构称为链队列。链队列的结构和各种基本操作均类似于线性链表，只是要注意它的插入和删除操作受限，只允许在队尾插入、队头删除。所以链队列其实就是只允许尾进头出的单链表。为了操作上的方便，我们将队头指针指向链队列的头结点，而队尾指针指向终端结点，如图 3.8 所示。

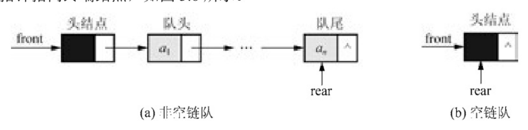


图 3.8　链队列示意图

链队列的结构定义如下。

```
typedef struct QNode            // 结点结构
{
    DataType data;
    struct QNode *next;
}QNode,*QueuePtr;

typedef struct                  //队列的链表结构
{
    QueuePtr  front,rear;       // 队头、队尾指针
}LinkQueue;
```

2. 链队列的操作实现

1) 链队的入队

入队操作就是在链表的尾部插入结点，如图 3.9 所示。

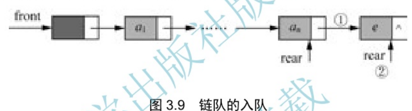


图 3.9 链队的入队

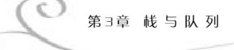
设插入元素 e 为 Q 的新队尾元素，则该链队的入队操作代码如下。

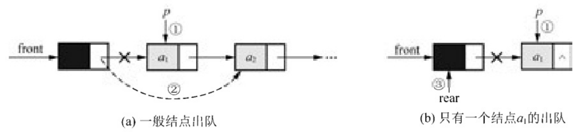
算法 3.23 链队的入队操作。

```
int EnQueue(LinkQueue *Q,int e)
{
  Q->rear->next=(QueuePtr)malloc(sizeof(QNode)); //①产生新节点,用 Q->rear
->next 来指向
  if(!Q->rear->next)
   { printf("存储分配失败!\n");
    return FALSE;
   }
  Q->rear=Q->rear->next;                     //②将 Q->rear 移到新节点
  Q->rear->data=e;
  Q->rear->next=NULL;
  return TRUE;
}
```

2) 链队的出队

出队操作就是头结点的后继结点出队，将头结点的后继改为它后面的结点，如图 3.10 所示。





(a) 一般结点出队　　(b) 只有一个结点a_1的出队

图 3.10　链队的出队

当链表除头结点外只剩下一个元素时，则需将 rear 指向头结点，如图 3.10(b)所示。若队列不空，则删除 *Q* 的队头元素，用 *e* 返回其值；并返回 TRUE；否则，返回 FALSE，如图 3.10(a)所示。其相应的代码实现如下：

算法 3.24　链队的出队操作。

```
int DeQueue(LinkQueue *Q, int *e)
{
    QueuePtr p;
    if(Q->front==Q->rear)
     {  printf("队列空! \n");
       return FALSE;
     }
    p=Q->front->next;          // ①将欲删除的队头结点暂存给 p
    *e=p->data;                // 将欲删除的队头结点的值赋值给 e
    Q->front->next=p->next;    // ②将原队头结点的后继 p->next 赋值给头结点后继
    if(Q->rear==p)             // ③若队头就是队尾,则删除后将 rear 指向头结点
        Q->rear=Q->front;
    free(p);
    return TRUE;
```

3) 显示元素

从队头到队尾依次输出队列 *Q* 中每个元素的代码如下。

算法 3.25　显示链队的元素。

```
int QueueTraverse(LinkQueue Q)
{
    QueuePtr p;
    if(Q.front==Q.rear)
    { printf("队列空!\n");
     return FALSE;
    }
    p=Q.front->next;
    while(p)
    {
        printf("%d ",p->data);
        p=p->next;
    }
    printf("\n");
    return TRUE;
}
```

对于循环队列与链队列的比较，可以从以下两方面来考虑。

(1) 从时间上说，它们的基本操作都是常数时间，即都是 $O(1)$，但是循环队列是事先申请好空间，使用期间不释放，而链队列每次申请和释放结点也会存在一些时间开销，如果入队出队频繁，则两者还是有细微差异的。

(2) 从空间上来说，循环队列必须有一个固定的长度，所以就有了存储元素个数和空间浪费的问题。而链队列不存在这个问题，尽管它需要一个指针域，会产生一些空间上的开销，但也可以接受。所以在空间上，链队列更加灵活。

总的来说，在可以确定长度最大值的情况下，建议用循环队列，如果无法预估队列的长度，则建议用链队列。

3.2.5 用队列实现银行排队叫号系统

下面用队列实现银行排队叫号系统，其主要包括排队管理、呼叫客户、打印客户信息、查找客户、删除客户等功能。

(1) 用链队列存储客户信息的定义如下。

```
typedef struct QNode
{
    char data[10];            // 客户名字
    struct QNode *next;       //下一个结点

} QNode;                      //链队结点类型

typedef struct{
    QNode *front,*rear;
}LinkQueue;  //客户队列的链表结构
```

(2) 主程序和各函数定义如下。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define false 0
//初始化队列,置为空
LinkQueue* InitQueue()
{
    LinkQueue *queue;
    QNode *node;
    queue=(LinkQueue*)malloc(sizeof(LinkQueue));//为队列头指针分配空间
    node=(QNode*)malloc(sizeof(QNode));      //为头结点分配空间
    node->next=NULL;
    queue->front=queue->rear=node;            //令队首和队尾指针均指向头结点
    return queue;
}
```

```
//入队操作,通过 strcpy()函数进行复制,在队尾插入姓名
void  InQueue(LinkQueue *Q, char  *x)
{
    QNode *p;
    p=(QNode *)malloc(sizeof(QNode));    //创建新结点 p
    if(p == NULL)
    {
        printf("内存不足");
        exit(1);
    }
    strcpy(p->data, x);                //将客户姓名存入新结点的数据域 data 中
    p->next=NULL;                      //将结点 p 的 next 域置为空
    Q->rear->next=p;                   //将新结点 p 入队
    Q->rear=p;                         //修改队尾指针
}

//通过 strcpy()函数进行复制,并通过指针在队头插入客户名
void  HeadInQueue(LinkQueue *Q,const  char  *x)
{
    QNode *p;
    p = (QNode *)malloc(sizeof(QNode));
    strcpy(p->data, x);            //创建新结点并将客户名存入数据域 data 中
    p->next = Q->front->next;      //修改 front 指针,在队头插入
    Q->front->next = p;
}

//出队操作
int OutQueue(LinkQueue *Q,char *x)
{
    QNode *p;
    if (Q->front==Q->rear) return 0;     //队空的情况
    p=Q->front->next;                    //令 p 指向队头元素
    strcpy(x, p->data);                  //读取队头元素,并保存到 x 中
    Q->front->next=p->next;              //出队
    if (p->next==NULL)                   //最后一个元素出队的情况
        Q->rear=Q->front;
    free(p);                             //释放空间
    return 1;
}

 //判定队列是否为空,不是空则把第一个删除,即办理第一个元素
int QueueEmpty(LinkQueue *Q)
{
```

```
    if (Q->front==Q->rear)
        return 1;                    //队空,返回 1
    else
        return 0;                    //队不空,返回 0
}

//通过 while()循环将队列输出,显示队列内容
void QueueTraverse(LinkQueue *Q)
{
    QNode *p = Q->front->next;       //令 p 指向队头元素
    while (p != NULL)
    {
        printf("%s ",p->data);       //显示当前元素的内容
        p=p->next;                   //令 p 指向下一个元素
    }
    printf("\n");
}

//通过 while()循环将队列元素与输入姓名进行比较,将查到的元素的位置输出
void  Search(LinkQueue *Q,  char  x[])
{
    QNode *p;
    int i=0;
    p=Qc->front->next;             //令 p 指向队头元素
    while ( strcmp(p->data,x )!=0 && p->next!=NULL) //比较 x 和当前元素的
内容是否相等
    {
        p=p->next;                 //令 p 指向下一个元素
        i++;                       //改变位置 i 的值,加 1
    }
    if ( strcmp(p->data,x )!=0 )
        printf("没有这个人\n");        //遍历完队列,没有找到 x
    else
        printf("前边有%d 人等待\n",i); //找到 x,并显示前面等待的人数 i
    free(p);
}

//通过 while()循环将队列遍历,将输入姓名的元素通过指针删除
int  Delete(LinkQueue *Q,char  x[])
{
    QNode *p,*q;
    int bfound = false;
    p = Q->front->next;              //初始时,p 指向队头元素
```

```
        q = Q->front;                           //初始时,q指向头结点
        while (p != NULL)                       //当前结点存在
        {
            if(strcmp(p->data, x ) == 0)        //在队列中找到元素x
            {
                q->next = p->next;              //令q指向p的下一个结点
                free(p);                        //释放当前结点
                p = NULL;
                return 1;                       //返回删除成功的信息
            }
            else
            {
                q = p;
                p = p->next;
            }
        }
    }

    //以菜单方式显示功能列表,从终端读取输入,执行对应的功能
    void main()
    {
        char sel,x;
        char name[10];
        LinkQueue *Q=InitQueue();  //初始化队列
        while (1)
        {
            printf("\n*********************************************\n");
            printf("   1.   排 队 管  理\n");
            printf("   2.   呼 叫 客  户\n");
            printf("   3.   打印客户信息\n");
            printf("   4.   查 找 客  户\n");
            printf("   5.   删 除 客  户\n");
            printf("   0.   清 空 队  列\n");
            printf("*********************************************\n 请选择
(操作的序号): ");

            scanf("%c",&sel); _flushall();
            switch(sel)
            {
            case '0':
                if (!QueueEmpty(Q))  //队不空
                    printf("请排队的客户明天再来\n");
                return;
```

```
            case '1':
                printf("输入客户姓名: ");
                scanf("%s",name);
                _flushall();
                printf("是否优先? (y/n): ");
                scanf("%c",&x);
                getchar();
                if(x=='y')  /*读入优先级,若是优先者,则调用 HeadInQueue 函数;否则
调用 InQueue 函数*/
                    HeadInQueue(Q,name);      //插入队头
                else
                    InQueue(Q,name);          //入队,插入队尾
                break;

            case '2':
                if (!OutQueue(Q,name))        //出队
                    printf("没有排队的客户\n");
                else
                    printf("请客户%s 办理\n",name);
                break;

            case '3':
                if (QueueEmpty(Q))            //队空
                {
                    printf("没有排队的客户\n");
                }
                else
                {
                    printf("排队者:");
                    QueueTraverse(Q);
                }
                break;

            case '4':
                printf("输入客户姓名: ");
                scanf("%s",name);
                 flushall();
                Search(Q,name);
                break;

            case'5':
                printf("输入客户姓名: ");
```

```
                scanf("%s",name);
                _flushall();
                if (Delete(Q,name)==1)
                    printf("已取消\n");
                else
                    printf("未找到该人\n");
                break;

            default:
                printf("选择错误,请重新选择。\n");
                break;
            }
        }
    }
```



独立实践

上例中优先级高的用户将直接加到队列的前面，因此只用了一个队列，请试着用两个队列来实现优先功能，即优先的客户进优先队列，正常的客户进正常队列。

小　　结

本章主要介绍栈与队列的基本概念、顺序存储表示、链式存储表示与基本操作。

栈是仅在表的一端进行插入和删除运算的线性表，又称为后进先出表(LIFO 表)。插入、删除端称为栈顶，另一端称为栈底。表中无元素称为空栈，当栈满时，做入栈运算必定产生空间溢出，称为“上溢”。当栈空时，做出栈运算必定产生空间溢出，称“下溢”。上溢是一种错误应设法避免，下溢常用作程序控制转移的条件。

队列是一种运算受限的线性表，允许删除的一端称为队首，允许插入的一端称为队尾。队列又称为先进先出线性表(FIFO 表)。顺序队列中存在“假上溢”现象，入队和出队操作使头尾指针只增不减，导致被删元素的空间无法利用，队尾指针超过向量空间的上界而不能入队。为克服“假上溢”现象，将向量空间想象为首尾相连的循环向量，存储在其中的队列称为循环队列。

习　　题

一、填空题

1．栈是________的线性表，其运算遵循________的原则。

2．当两个栈共享一个存储区时，栈利用一维数组 s[N]表示，两栈顶指针为 top1 与 top2(栈顶指针均指向当前栈顶元素所在单元的位置)，则当栈 1 空时，top1 为________，栈 2 空时，top2 为________，栈满时为________。

3．用 S 表示入栈操作，X 表示出栈操作，若元素入栈的顺序为 1234，为了得到 1342 出栈顺序，相应的 S 和 X 的操作串为________。

4．顺序栈用 data[N]存储数据，栈顶指针是 top(栈顶指针指向当前栈顶元素所在单元的位置)，则值为 x 的元素入栈的操作是________。

5．用下标 0 开始的 N 元数组实现循环队列时，为实现下标变量 M 加 1 后在数组有效下标范围内循环，可采用的表达式是 M=________。

6．已知链队的头尾指针分别是 f 和 r，则将值 x 入队的操作序列是________。

7．区分循环队列的满与空有两种方法，它们是________和________。

8．循环队列引入的目的是克服________。

9．一个栈的输入序列是 1，2，3，则不可能的栈输出序列是________。

10．队列是插入只能在表的一端，而删除在表的另一端进行的线性表，其特点是________。

二、选择题

1．在做入栈运算时，应先判别栈是否(　①　)，在做出栈运算时应先判别栈是否(　②　)。当栈中元素为 n 个时，做入栈运算时发生上溢，则说明该栈的最大容量为(　③　)。为了增加内存空间的利用率和减少溢出的可能性，由两个栈共享一片连续的内存空间，将两栈的(　④　)分别设在这片内存空间的两端，这样，当(　⑤　)时，才会产生上溢。

①，②：A．空　B．满　C．上溢　D．下溢

③：A．$n-1$　B．n　C．$n+1$　D．$n/2$

④：A．长度　B．深度　C．栈顶　D．栈底

⑤：A．两个栈的栈顶同时到达栈空间的中心点

B．其中一个栈的栈顶到达栈空间的中心点

C．两个栈的栈顶在栈空间的某一位置相遇

D．两个栈均不空，且一个栈的栈顶到达另一个栈的栈底

2．一个栈的输入序列为 1，2，3，…，n，若输出序列的第一个元素是 n，输出第 $i(1\leqslant i\leqslant n)$个元素是(　　)。

A．不确定　B．$n-i+1$　C．i　D．$n-i$

3．若一个栈的输入序列为 1，2，3，…，n，输出序列的第一个元素是 i，则第 j 个输出元素是(　　)。

A．$i-j-1$　B．$i-j$　C．$i-j+1$　D．不确定的

4．有六个元素 7，6，5，4，3，2，1，从 7 开始顺序进栈，则下列(　　)不是合法的出栈序列。

A．7 5 4 3 6 1 2　B．4 5 3 1 2 6 7　C．7 3 4 6 5 2 1　D．2 3 4 1 5 6 7

5．设一个栈的输入序列是 1，2，3，4，5，则下列序列中，栈的合法输出序列的是(　　)。

A．5 1 2 3 4　B．4 5 1 3 2　C．4 3 1 2 5　D．3 2 1 5 4

6．若栈采用顺序存储方式存储，现两栈共享空间 $V[m]$，top[i]代表第 i 个栈(i=1，2)栈顶，栈 1 的底在 v[0]，栈 2 的底在 $V[m-1]$，则栈满的条件是(　　)。

A．top[2]−top[1]=0　B．top[1]+1=top[2]

C．top[1]+top[2]=m　D．top[1]=top[2]

7．执行完下列语句段后，i 值为（　　）．

```
int f(int x)
{ return ((x>0) ? x* f(x-1):2); }
int i;
i =f(f(1));
```

A．2　　B．4　　C．8　　D．无限递归

8．用链接方式存储的队列(不带头结点)，在进行删除运算时(　　)。

A．仅修改头指针　　B．仅修改尾指针

C．头、尾指针都要修改　　D．头、尾指针可能都要修改

9．假设以数组 $A[m]$存放循环队列的元素，其头尾指针分别为 front 和 rear，则当前队列中的元素个数为(　　)。

A．(rear−front+m)%m　　B．rear−front+1

C．(front−rear+m)%m　　D．(rear-front)%m

10．若用一个大小为 6 的数组来实现循环队列，且当前 rear 和 front 的值分别为 0 和 3，当从队列中删除一个元素，再加入两个元素后，rear 和 front 的值分别为(　　)。

A．1 和 5　　B．2 和 4　　C．4 和 2　　D．5 和 1

11．设栈 S 和队列 Q 的初始状态为空，元素 e_1，e_2，e_3，e_4，e_5 和 e_6 依次通过栈 S，一个元素出栈后即进入队列 Q，若 6 个元素出队的序列是 e_2，e_4，e_3，e_6，e_5，e_1，则栈 S 的容量至少应该是(　　)。

A．6　　B．4　　C．3　　D．2

三、简答题

1．什么是栈？什么是队列？它们各自的特点是什么？

2．线性表、栈、队列有什么异同？

3．简述栈的入栈、出栈操作的过程。

4．在循环队列中简述入队、出队操作的过程。

5．在什么情况下，才能使用栈、队列等数据结构？

四、应用题

1．设计算法判断一个算术表达式的圆括号是否正确配对。(提示：对表达式进行扫描，凡遇‘(’就进栈，遇‘)’就退掉栈项的‘(’，表达式扫描完毕，栈应为空，否则，圆括号不配对。)

2．假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点(不设头指针)，请编写相应的置空队、入队列和出队列的算法。

3．在某程序中，有两个栈共享一个一维数组空间 SPACE[N]，SPACE[0]、SPACE[N-1]分别是两个栈的栈底。

(1) 对栈 1、栈 2，试分别写出(元素 x)入栈的主要语句和出栈的主要语句。

(2) 对栈 1、栈 2，试分别写出栈满、栈空的条件。

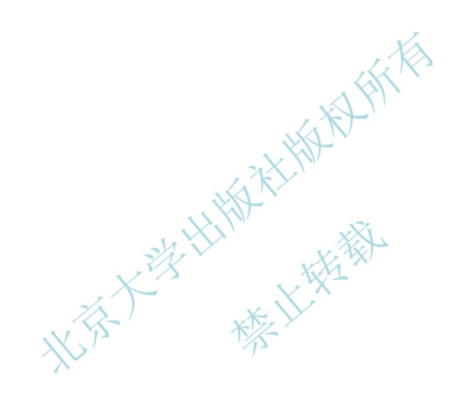
4．如果用一个循环数组 q[m]表示队列时，该队列只有一个队列头指针 front，不设队列尾指针 rear，而改置计数器 count 用以记录队列中结点的个数。

(1) 编写实现队列判空、入队、出队的三个基本运算算法。

(2) 队列中能容纳元素的最多个数是多少？

5．设输入元素为 1、2、3、P 和 A，输入次序为 123PA。当所有元素均到达输出序列后，有哪些序列可以作为高级语言的变量名。

6．假设以数组 Sq[MAX]存放循环队列的元素，同时设变量 rear 和 len 分别指示循环队列中队尾元素的位置和内含元素的个数。请给出判别此循环队列的队满条件，并写出相应的入队列和出队列的算法。





串

问题描述

字符串分析

在程序开发中，经常碰到各类字符串(以下简称串)问题，如在很多应用程序中经常要传输用户注册的相关信息，当用户把数据提交到服务器上时通常使用的就是串操作。这时需要对所输入的字符串进行分析。

4.1　串的类型与基本运算

4.1.1　串的类型定义

串是由零个或多个字符组成的有限序列。一般记作：

$$s="c_0c_1c_2\ldots c_{n-1}" \qquad (n\geqslant 0)$$

其中，s 为串名，用双引号括起来的字符序列是串的值；$c_i\,(0\leqslant i\leqslant n-1)$可以是字母、数字或其他字符；双引号为串值的定界符，不是串的一部分；串字符的数目 n 称为串的长度。

零个字符的串称为空串，通常以两个相邻的双引号来表示空串，如 s=""，它的长度为零。

注意：要区分空串和空格串(也称空白串)，空格串仅由空格组成，如 s=" "，长度不为零。

串中任意个连续字符组成的序列称为该串的子串，包含子串的串被称为主串，通常将子串在主串中首次出现时的该子串的首字符对应的主串中的序号，定义为子串在主串中的序号(或位置)。

例如，有两个串 A 和 B。

A="This is a string."

B="is"

显然，B 是 A 的子串，B 在 A 中的序号是 3 而不是 6。

特别地，空串是任意串的子串，任意串是其自身的子串。

当且仅当两个串的值相等时称这两个串是相等的。也就是说，只有当两个串的长度相等，并且各个对应位置的字符都相等时这两个串才相等。

4.1.2　串的基本运算

串常用的基本运算主要包括如下几种。

假设用大写字母 S、T 等表示串，用小写字母表示组成串的字符，并且假设：

```
S1=″a1a2……an″
S2=″b1b2……bm″
```

(1) 赋值(StrAssign)

StrAssign(S，*chars)给串 s 赋值，其中*chars 可为串变量、串常量或经过适当运算所得到的串值，生成一个其值等于字符串常量 chars 的串 S。例如：

```
StrAssign(S, ″abc″);
StrAssign(S, S1);
StrAssign(S, ″ ″);
```

(2) 联接(Strcat)

Strcat(S_1，S_2)就是将串 S_2 紧接着放在串 S_1 的串值的末尾，组成一个新的串 S_1。

例如：

```
Strcat(S1,S2);
```

则 S_1="$a_1a_2\cdots a_nb_1b_2\cdots b_m$"

请思考 Strcat(s_2，s_1)的结果，两者相同吗？

(3) 求串长(Strlen)

Strlen(S)表示求串 S 的长度，它是一个整型函数。

例如：

```
Strlen(S1)=n
Strlen(″abc″)=3
Strlen(″ ″)=0
```

(4) 求子串(Substr)

Substr(S，pos，len，Sub)表示在串 S 中截取从第 pos 个字符开始的连续 len 个字符，构成一个新串 Sub。其中参数应满足：

```
1≤pos≤Strlen(S),1≤len≤Strlen(S)-pos+1
```

例如：

```
SubStr ("abcd",2,2,Sub);
SubStr ("a1a2…an",i,j,Sub);
```

有时对参数 len 的限制可以放宽到 len≥0，当 len=0，规定对任何串 S，有 Substr (S，pos，0，Sub)，得到 Sub=" "；当 len＞Strlen(S)−pos+1 时，规定取 S 的第 pos 个字符直到 S 的最后一个字符作为子串，该子串共有 Strlen(S)−pos+1 个字符。

(5) 比较串的大小(Strcmp)

Strcmp(S，T)是一个函数，它的功能是比较两个串 S 和 T 的大小，其中函数值小于、等于和大于 0 时，分别表示 S＜T、S=T 和 S＞T。

串中可能出现的字符，依赖于 ASCII 码及国标码的字符集，字符的大小是由该字符在字符集中出现的先后次序确定。若用函数 ord 表示字符在字符集中的序号，当 ord(ch1)＜ord(ch2)时，则 ch1＜ch2。常用的字符集都规定：数字字符 0，1，…，9 在字符集中是顺序排列的，字母字符 A，B，……，Z(或者 a，b，…，z)在字符集中也是顺序排列的。因此有：

```
ord('a')<ord('b')…<ord('z')
ord('0')<ord('1')…<ord('9')
```

定义了字符的大小之后，就可以定义两个串的大小。串的大小通常是按字典序定义的，即从两个串的第一个字符起，逐个比较相应的的字符，直到找到两个不等的字符为止，这两个不等的字符即可确定串的大小。例如，"this"＞"there"，这是因为'i'＞'e'。若找不到两个不等的字符，就必须由串长来决定大小。例如"there"＞"the"，这是因为两个串的前三个字符均一一相同。但前者长度大于后者。由此可知，两个串相等当且仅当其长度相等，以及各个对应位置上的字符也相同。

(6) 插入(StrInsert)

StrInsert(s，pos，t)表示将串 t 插入在串 s 的位置 pos 上。例如：执行 StrInsert(S_1，2，"abc")后：

```
S1="a1a2abca3…an"
```

(7) 删除(StrDelete)

StrDelete(S，pos，len)表示从串 s 中删除第 pos 个字符开始的连续 len 个字符。例如，StrDelete(S_1，3，n-3)执行结果为：

```
S1="a1a2an"
```

(8) 子串定位(Index)

Index(S，T)是一个求子串在主串中位置的定位函数，它表示在主串 S 中查找是否有等于 T 的子串，若有，则函数值为 T 在 S 中首次出现的位置；若无，则函数值为零。

例如：

```
Index("abcdbc","bc")=2
Index("abcdbc","ac")=0
```

有时，子串定位函数也用 Index(S，T，pos)，表示返回子串 T 在主串 S 中第 pos 个字符之后的位置。若不存在，则返回零。

例如：

```
Index("This is a book!","is",4)=6
```

(9) 置换(Replace)

Replace(S，T，V)表示用 V 替换所有在 S 中出现的、和 T 相等的子串。例如：设 S="abcdbca"，T="bc"，V="¢"则执行后：

```
S="a ¢ d ¢ a"
```

上述串运算都是基本的，它就像数值计算中对整型变量进行四则运算那样，频繁地用在串处理中，因此，在引进串变量的高级语言中，一般都作为基本运算符或基本内部函数来提供，当然提供的种类和符号在各个语言可能有所不同。

4.2 串 的 存 储

由于串实际上是一种特殊的线性表，它的元素仅由一个字符组成，因此串的存储方法也就是线性表的一般方法，常见的有顺序存储和链式存储两种方法。

4.2.1 串的顺序存储

和线性表的顺序存储一样，串的顺序存储结构就是采用与其逻辑结构相对应的存储结构，即串的各个字符按顺序存入连续的存储单元，逻辑上相邻的字符在内存中也是相邻的，有时称为顺序串。

串的最简单的程序存储方式是采用非紧缩格式，即每一个存储单元中存放一个字节，所占存储单元数目即为串的长度。这种存储结构，随机读/写串中指定的第 i 个字符最为方便，存取的速度最快。但每一个存储单元本可以放下多个字符，只放一个字符不能充分利用存储空间。

为了充分利用存储空间，可以采用紧缩格式的顺序存储结构，即根据存储单元的容量给每个单元存入多个字符，最末一个单元如果没有占满，则可填充空格符。这种存储结构从所占存储单元的数目不能求出准确的串长度(末尾单元可能是空余单元)，故需要对串的长度进行设定。

在串的顺序存储结构中，表示串的长度通常有两种方法：一种方法是设置一个串的长度参数，此种方法的优点是便于在算法中用长度参数控制循环过程；另一种方法是在串值的末尾添加结束标记'\0'，此种方法的优点是便于系统自动实现。

例如，定义一个串变量 String s，则这种存储方式可以直接得到串的实际长度，即 s.length，如图 4.1 所示。

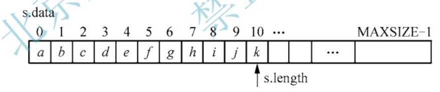


图 4.1 串的顺序存储方式 1

在串尾存储一个不会在串中出现的特殊字符作为串的终止符，以此表示串的结尾。如图 4.2 所示，它是用'\0'来表示串的结束。这种存储方法不能直接得到串的长度，它使用判断当前字符是否是'\0'来确定串是否结束，从而求得串的长度。

char s[MAXSIZE]

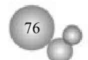
0	1	2	3	4	5	6	7	8	9	10	…			MAXSIZE−1
a	*d*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	\0		…	

图 4.2 串的顺序存储方式 2

串的顺序存储结构就是用数组存放串的所有字符，数组有静态数组和动态数组两种。

1. 静态数组结构

静态数组结构也称为定长顺序结构，其数据结构类型定义如下。



```
typedef struct{
    char str[MAXSIZE];
    int length;
}String;
```

2. *动态数组结构*

动态数组结构也称堆分配存储结构。为了实现动态存储，我们定义动态数组的结构体如下。

```
typedef struct
{
    char * str;
    int maxLength;
    int length;
} DString;
```

其定义中比静态数组结构体增加了一个指出动态数组长度的域，为每次动态存储空间提供增加量。同时，动态数组下串的基本操作增加了初始化操作和撤销操作。

4.2.2　串的链式存储

串的链式存储结构与线性表的链式存储结构类似，是将存储区分成许多“结点”，每个结点包含一个存放字符的域和一个存放指向下一个结点的指针域。采用链式存储结构的串称为链串。由于串的特殊性——每个元素只包含一个字符，因此，每个结点可以存放一个字符，也可以存放多个字符。图 4.3 和图 4.4 分别表示了存储密度为 4 和 1 的链式存储结构。

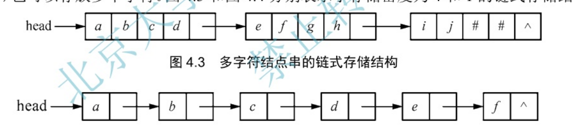


图 4.3　多字符结点串的链式存储结构

图 4.4　单字符结点串的链式存储结构

当结点大小大于 1(如结点大小等于 4)时，链串的最后一个结点的各个数据域不一定总能全被字符占满。此时，应在这些未占用的数据域里补上不属于字符集的特殊符号(如“#”字符)，以示区别，如图 4.3 中的最后一个结点所示。

为了实现链串的基本操作，串的链式存储结构类型描述如下。

```
typedef struct node
{
   char data;
   struct node *next;
}LinkStrNode;
typedef LinkStrNode * LinkString;    //链式串的指针
```

为了便于进行串的操作，当以链表存储串值时，除头指针外还可附设一个尾指针指示链表中的最后一个结点，并给出当前串的长度。我们称如此定义的串存储结构为块链结构，

则串的块链结构类型可描述如下。

```
#define SIZE 60                              //可由用户定义的块大小
typedef struct ChunkNode
{
    char data[SIZE];
 struct ChunkNode *next;
}ChunkNode;

typedef struct
{
  int curlen;                               //块链串的当前长度
  ChunkNode *head,*tail;                    //块链串的头和尾指针
} ChunkLinkStr;

typedef ChunkLinkStr * ChunkLinkString; //块链串的指针
```

在链式存储方式中，结点大小的选择和顺序存储方式的格式选择一样重要，它直接影响着串处理的效率。在各种的串的处理系统中，所处理的串往往很长或很多，例如，一本书的几百万个字符，情报资料的成千上万个条目。这要求我们考虑串值的存储密度。存储密度可定义为

$$存储密度=\frac{串值所占的存储位}{实际分配的存储位} \tag{4.1}$$

显然，存储密度小(如结点大小为 1)时，运算处理方便，然而，存储占用量大。如果在串处理过程中需要进行内、外存交换，则会因为内外存交换操作过多而影响处理的总效率。应该看到，串的字符集大小也是一个重要因素。一般地，字符集小，则字符的机内编码短，这也影响串值的存储方式的选取。

4.3* 串的模式匹配

串的模式匹配即子串定位是一种重要的串的运算。设 S 和 T 是给定的两个串，在主串 S 中找到等于子串 T 的过程称为模式匹配(其中 T 称为模式串)，如果在 S 中找到 T 的子串，则称匹配成功，函数返回 T 在 S 中首次出现的存储位置(或序号)，否则匹配失败，返回-1。

为了运算方便，设字符串的长度存放在 0 号单元，串值从 1 号单元存放，这样字符序号与存储位置一致。本节主要介绍模式匹配的简单算法和一种改进的算法(KMP 算法)。

4.3.1 模式匹配的简单算法

其基本思想是分别利用计数指针 i 和 j 指示主串 S 和模式串 T 中当前正待比较的位置。首先将 $S[1]$和 $T[1]$进行比较，若不同，则将 $S[2]$和 $T[1]$进行比较，……，直到 S 中的某一个字符 $S[i]$和 $T[1]$相同，再将它们之后的字符进行比较，若也相同，则如此继续向下比较，当 S 的某一个字符 $S[i]$与 T 的字符 $T[j]$不同时，则 S 返回到本趟开始字符的下一个字符，即 $S[i-j+2]$，T 返回到 $T[1]$，继续开始下一趟的比较，重复上述过程。若 T 中的字符全部比较完，则说明本趟匹配成功，否则，匹配失败。

设主串 S="ababcabcacbab"，模式串 T="abcac"，匹配过程如图 4.5 所示。

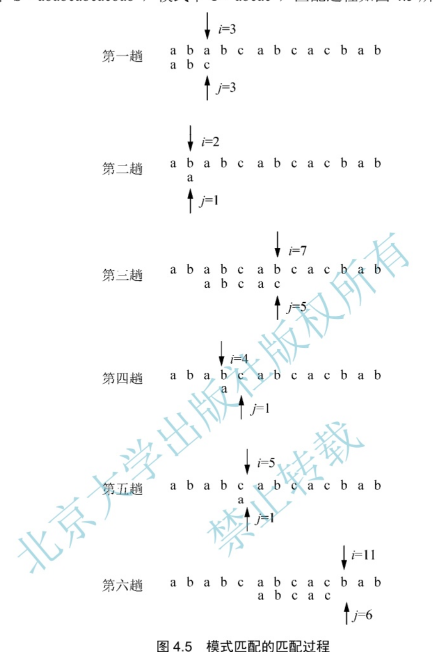


图 4.5　模式匹配的匹配过程

其算法执行步骤如下。

(1) 判断匹配位置是否到串的末尾。

(2) 如果主串与模式串对应字符相等，则继续匹配下一字符。

(3) 否则主串、子串指针回溯则重新开始下一次匹配数据元素。

(4) 判断是否匹配成功。

前面我们已经用串的其他操作实现了模式匹配的算法 Index()。现在考虑不用串的其他操作，而是用基本的数组来实现同样的算法。注意，这里假设主串 S 和要匹配的子串 T 的长度都存在 $S[0]$和 $T[0]$中，对应的代码见算法 4.1。其中 i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开始匹配，j 是用于子串 T 中当前位置下标值，当 i 小于 S 的长度并且 j 小于 T 的长度时，循环继续。

算法 4.1 模式匹配的简单算法。

```
int Index(char S[], char T[], int pos) {
    int i = pos;
    int j = 1;
    while (i <= S[0] && j <= T[0])        //0 号单元存放串的长度
    {
        if (S[i] == T[j])
        {
            ++i;
            ++j;
        }
        else
        {
            i = i-j+2;                    //i 退回到上次匹配首位的下一位
                j = 1;                    //j 退回到子串 T 的首位
        }
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}
```

下面分析它的时间复杂度，设串 S 长度为 n，串 T 长度为 m。匹配成功的情况下考虑两种极端情况。

(1) 在最好的情况下，每趟不成功的匹配都发生在第一对字符比较时。

例如，S="aaaaabc"，T="bc"，设匹配成功发生在 $S[i]$处，则字符比较次数在前面 i-1 趟中共比较了 i-1 次，第 i 趟成功的匹配共比较了 m 次，所以总共比较了 i-1+m 次，所有匹配成功的可能共有 n−m+1 种，设从 $S[i]$开始与 T 串匹配成功的概率为 p_i，在等概率情况下 $p_i=1/(n-m+1)$，因此最好情况下平均比较的次数是

$$\sum_{i=1}^{n-m+1} p_i \times (i-1+m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i-1+m) = \frac{n+m}{2} \tag{4.2}$$

即最好的情况下的时间复杂度是 $O(n+m)$。

(2) 在最坏的情况下，每趟不成功的匹配都发生在 T 的最后一个字符。

例如，当 S="aaaaaaaaaabc"，T="aaab"时，设匹配成功发生在 $S[i]$处，则在前面 i-1 趟匹配中共比较了(i-1)×m 次，第 i 趟成功的匹配共比较了 m 次，所以共比较了 $i\times m$ 次，因此最坏情况下平均比较次数是

$$\sum_{i=1}^{n-m+1} p_i \times (i \times m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i \times m) = \frac{m \times (n-m+2)}{2} \tag{4.3}$$

即最坏情况下的时间复杂度是 $O(n \times m)$ 。在 4.3.2 节，我们将介绍另一种较好的模式匹配算法。

当模式串为"00000001"，而主串为"0000000000000000000000000000000000000000000000000000"时，由于模式中前 7 个字符均为“0”，主串中前 52 个字符也均为“0”，每趟

比较都在模式的最后一个字符出现不等时，将指针回溯 i-6 的位置上，并从模式的第一个字符开始重新比较，整个匹配过程中指针 i 需要回溯 45 次，则 while 循环次数为 46*8(index*m)。可见，算法 4.1 在最坏的情况下的时间复杂度为$O(n\times m)$ 。这种情况在只有 0、1 两个字符的文本串处理中经常出现。01 串可以用在许多应用之中，比如一些计算机的图形显示就是用画面表示的一个 01 串，一页书就是一个几百万个 0 和 1 组成的串。在二进位计算机上实际处理的都是 01 串，一个字符的 ASCII 码也可以看作八个二进位的 01 串，而汉字存储在计算机中处理时也是作为一个 01 串和其他的字符串一样看待的。因此，我们有必要介绍一种改进的模式匹配算法。

4.3.2 KMP 算法

这种算法是由 D. E. Knuth 与 V. R. Pratt 和 J. H. Morris 同时发现的，因此人们称它为克努特-莫里斯-普拉特操作(简称 KMP 算法)。此算法可以在 $O(n+m)$的时间数量级上完成串的模式匹配操作。其改进在于：每当一趟匹配过程中出现字符比较不等时，不需回溯 i 指针，而是利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离后，继续进行比较。下面先从具体例子看起。

回顾图 4.5 中的匹配过程示例，在第三趟的匹配中，当 i=7、j=5 字符比较不等时，又从 i=4、j=1 重新开始比较。然而，经仔细观察发现，在 i=4 和 j=1，i=5 和 j=1，i=6 和 j=1 这三次比较都不必进行的。因为从第三趟部分匹配的结果就可得出，主串中第 4～6 个字符必然是'b'、'c'和'a' (即模式串第 2～4 个字符)。因为模式中的第一个字符是 a，因此它无需再和这三个字符进行比较，而仅需将模式串向右滑动三个字符的位置进行 i=7、j=2 时的字符比较即可。同理，在第一趟匹配中出现字符不等时，仅需将模式串向右移动两个字符的位置继续进行 i=3、j=1 时的字符比较。由此，在整个匹配过程中，i 指针没有回溯，如图 4.6 所示。

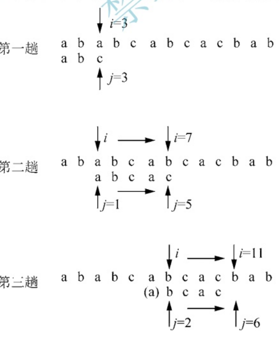


图 4.6 改进算法的匹配过程示例

当主串中第 i 个字符与模式中的第 j 个字符“失配”(即比较不相等)时，主串中第 i 个字符(i 指针不回溯)应与模式中哪个字符比较呢？

假设应与模式中第 $k(k<j)$个字符继续比较，则模式中前 k−1 个字符的子串必须满足下列关系

$$'p_1p_2\cdots p_{k-1}'='s_{i-k+1}s_{i-k+2}\cdots s_{i-1}' \tag{4.4}$$

而已经得到的匹配结果是

$$'p_{j-k+1}p_{j-k+2}\cdots p_{j-1}'='s_{i-k+1}s_{i-k+2}\cdots s_{i-1}' \tag{4.5}$$

由式 4.3 和式 4.4 推得下列等式

$$'p_1p_2\cdots p_{k-1}'='p_{j-k+1}p_{j-k+2}\cdots \mathrm{p}_{j-1}' \tag{4.6}$$

反之，若模式串中存在满足式(4.5)的两个子串，则匹配过程中，主串中第 i 个字符与模式中第 j 个字符比较不等时，仅需将模式向右滑动至模式中第 k 个字符与主串中第 i 个字符对齐，此时，模式中头 k−1 个字符的子串 $'p_1p_2\cdots p_{k-1}'$ 必定与主串中第 i 个字符之前长度为 k−1 的子串 $'s_{i-k+1}s_{i-k+2}\cdots s_{i-1}'$ 相等，由此，匹配仅需从模式中第 k 个字符与主串中第 i 个字符比较起继续进行。

若令 next[j]=k，则 next[j] 表明当前模式中第 j 个字符与主串中相应字符“失配”时，在模式中需重新和主串中该字符进行比较的字符位置。由此可引出模式串的 next 函数的定义。

$$\mathrm{next}[j]=\begin{cases}0 & j=1\\ \mathrm{Max}\{k \mid 1<k<j \text{ 且} 'p_1p_2\cdots p_{k-1}'='p_{j-k+1}\cdots p_{j-1}'\} & \\ 1 & \text{其他情况}\end{cases} \tag{4.6}$$

由定义可推出模式串的 next 函数值如下：

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

在求得模式的 next 函数之后，匹配可如下进行。

假设以指针 i 和 j 分别指示主串和模式中正待比较的字符，令 i 的初值为 pos，j 的初值为 1。若在匹配过程中 $s_i=p_i$，则 i 和 j 分别增 1，否则，i 不变，而 j 退到 next[j] 的位置再比较，若相等，则指针各自增 1，否则 j 再退到下一个 next 值的位置，以此类推，直到下列两种可能：一种是 j 退到某个 next 值(next[next[…next[j]…]])时字符比较相等，则指针各自增 1，继续进行匹配；另一种是 j 退到值为零(即模式的第一个字符“失配”)，则此时需将模式继续向右滑动一个位置，即主串的下一个字符 s_{i+1} 起和模式重新开始匹配。图 4.7 正是上述匹配过程的一个例子。

当匹配过程中产生“失配”时，指针 i 不变，指针 j 退回到 next[j] 所指示的位置上重新进行比较，当指针 j 退至零时，指针 i 和指针 j 需同时增 1，即若主串的第 i 个字符和模式中的第 1 个字符不等，则应从主串的第 i+1 个字符起重新进行匹配。对应 KMP 的实现见算法 4.2，其中 i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开始匹配；j 用于子串 T 中当前位置下标值。

```
                ↓ i=2
第一趟  主串  a c a b a a b a a b c a c a a b c
        模式  a b
                ↑ j=2    next[2]=1

                ↓ i=2
第二趟  主串  a c a b a a b a a b c a c a a b c
        模式    a
                ↑ j=1    next[1]=0

                  ↓ i=3 ──→ ↓ i=8
第三趟  主串  a c a b a a b a a b c a c a a b c
        模式      a b a a b c
                  ↑ j=1 ──→ j=6        next[6]=3

                        ↓ i=8 ──→ ↓ i=14
第四趟  主串  a c a b a a b a a b c a c a a b c
        模式          (a b) a a b c a c
                        ↑ j=3 ──→ ↑ j=9
```

图 4.7 利用模式的 next 函数进行匹配的过程示例

算法 4.2 模式匹配的 KMP 算法。

```
int Index_KMP(char S[], char T[], int pos){
    int i = pos;
    int j = 1;
    int next[100];
    GetNextValue(T, next);
    while (i <= S[0] && j <= T[0])
    {
        if (j==0 || S[i] == T[j])
        {
            ++i;
            ++j;
        }
        else                    //指针后退重新开始匹配
            j = next[j];        //j 退回到合适的位置,i 值不变
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}
```

该算法返回子串 T 在主串 S 中第 pos 个字符之后的位置。若不存在，则函数返回值为 0。其中 T 非空，pos 应该在[1，StrLength(S)]之间。

KMP 算法是在已知模式串的 next 函数值的基础上执行的，那么，如何求得模式串的 next 函数值呢？next 数组的求解方法是第一位的 next 值为 0，第二位的 next 值为 1，后面求解每一位的 next 值时，根据前一位进行比较。首先将前一位与其 next 值对应的内容进行比较，如果相等，则该位的 next 值就是前一位的 next 值加 1；如果不等，则向前继续寻找 next 值对应的内容来与前一位进行比较，直到找到某个位上内容的 next 值对应的内容与前一位相等为止，则这个位对应的值加 1 即为需求的 next 值；如果找到第一位仍然没有找到与前一位相等的内容，那么需求位上的 next 值即为 1。按上述模式串的 next 函数值

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

来对其具体运算的过程表述如下。

(1) 前两位必定为 0 和 1。

(2) 计算第三位的时候，要根据第二位 b 的 next 值决定，若为 1，则把 b 和 1 对应的 a 进行比较；若不同，则第三位 a 的 next 的值为 1，因为一直比较到最前一位，都没有发生比较相同的现象。

(3) 计算第四位的时候，要根据第三位 a 的 next 值决定，若为 1，则把 a 和 1 对应的 a 进行比较；若相同，则第四位 a 的 next 的值为第三位 a 的 next 值加上 1，即为 2。因为在第三位实现了其 next 值对应的值与第三位的值相同。

(4) 计算第五位的时候，要根据第四位 a 的 next 值决定，若为 2，则把 a 和 2 对应的 b 进行比较；若不同，则再将 b 对应的 next 值 1 对应的 a 与第四位的 a 进行比较；若相同，则第五位的 next 值为第二位 b 的 next 值加 1，即为 2。因为在第二位实现了其 next 值对应的值与第四位的值相同。

(5) 计算第六位的时候，要根据第五位 b 的 next 值决定，若为 2，则把 b 和 2 对应的 b 进行比较；若相同，则第六位 c 的 next 值为第五位 b 的 next 值加 1，即为 3，因为在第五位实现了其 next 值对应的值与第五位相同。

(6) 计算第七位的时候，要根据第六位 c 的 next 值决定，若为 3，则把 c 和 3 对应的 a 进行比较；若不同，则再把第 3 位 a 的 next 值 1 对应的 a 与第六位 c 比较；若仍然不同，则第七位的 next 值为 1。

(7) 计算第八位的时候，要根据第七位 a 的 next 值决定，若为 1，则把 a 和 1 对应的 a 进行比较；若相同，则第八位 c 的 next 值为第七位 a 的 next 值加上 1，即为 2，因为在第七位实现了其对应的值与第七位相同。

求 next 值的对应代码见算法 4.3。

算法 4.3 模式串的 next 函数

```
void GetNextValue(String T, int *next) {
    int i,j;
    i=1;
    j=0;
```

```
        next[1]=0;
        while (i<T[0])
        {
            if(j==0 || T[i]== T[j])
            {
                ++i;
                ++j;
                next[i] = j;
                }
            else
                j= next[j];                    //若字符不相同,则 j 值回溯
        }
    }
```

注意：虽然简单模式匹配算法的时间复杂度为 $O(n \times m)$，但在一般情况下，其实际的执行时间近似于 $O(n+m)$。KMP 算法仅当模式与主串之间存在许多"部分匹配"的情况下才显得比简单算法快得多。4.3.3 节介绍 KMP 模式匹配算法的改进。

4.3.3 KMP 模式匹配改进算法

前面定义的 next 函数在某些情况下尚有缺陷。例如，当模式"a a a a b"在和主串"a a a b a a a a b"匹配时，若 i=4、j=4，则 $S[4] \neq T[4]$，由 next[j] 的指示还需进行 i=4、j=3，i=4、j=2，i=4、j=1 这三次比较。实际上，因为模式中的第一、二、三个字符和第四个字符都相等，因此不需要再和主串中的第四个字符相比较，而可以将其向右滑动四个字符的位置直接进行 i=5、j=1 时的字符比较。这就是说，若按上述定义得到 nextval[k] = k ，而模式中 p_i=p_k，即此时的 next[j] 应和 next[k] 相同。由此可计算 next 函数的修正值 nextval，图 4.8 为上述模式串的 next 函数值和 nextval 函数值。

j	1	2	3	4	5
模式	a	a	a	a	b
next[j]	0	1	2	3	4
nextval[j]	0	0	0	0	4

图 4.8 模式串的 next 函数值和 nextval 函数值

改进后求 next 值的代码见算法 4.4，其中 $T[0]$表示串 T 的长度。

算法 4.4 模式匹配的 KMP 改进算法

```
    Void GetNextValueImp (String T, int *nextval) {
        int i,j;
        i=1;
        j=0;
        nextval[1]=0;
        while (i<T[0])
        {
            if(j==0 ||T[i]==T[j]) /*T[i]表示后缀的单个字符,T[j]表示前缀的单个字符*/
```

```
            {
                ++i;
                ++j;
                if (T[i]!=T[j])          // 若当前字符与前缀字符不同
                    nextval[i] = j;      // 则当前的 j 为 nextval 在 i 位置的值
                else
                    nextval[i] = nextval[j]; /*若当前字符与前缀字符相同,则将前
缀字符的
                                          /*nextval 值赋给 nextval 在 i 位置的值*/
        }
            else
                j= nextval[j];           //若字符不相同,则 j 值回溯
        }
    }
```

4.4 用串实现对字符串的分析

下面将用前面所学的知识实现对以下字符串的分析，要求将下面字符串中的编号、姓名、性别、电话、地址等信息解析成结构体，主要功能包括字符串的初始化、字符串解析成结构体、结构体输出为字符串等。

例如，待分析的字符 mystr://num=123|name=wwdd|sex=1|phone=40923456|addr=hangzhou

程序编写如下。

```
#include <stdio.h>
#include <string.h>
enum
{   SUCESS,
    PTRNULL,
    NODATA,
};
typedef struct
{
    char num[5];          //编号
    char name[9];         //姓名
    char sex[3];          //性别
    char phone[13];       //电话
    char addr[31];        //地址
}ElemType;

/***********************************/
/*      菜单选择函数程序          */
/***********************************/
int menu select()
{
    int sn;
```

```
    printf("==============================\n");
    printf("   字符串分析小系统\n");
    printf("==============================\n");
    printf("   1.字符串初始化\n");
    printf("   2.字符串分析成结构体\n");
    printf("   3.结构体输出为字符串\n");
    printf("   0.退出系统\n");
    printf("==============================\n");
    printf("请选择对应操作的序号：\n");
    for(;;)
    {
        scanf(" %d",&sn);
        if(sn < 0||sn > 4)
            printf("\n\t 输入错误,重选 0~4");
        else
            break;
    }
    return sn;
}
int InitString(char *pStr);
int String2Struct(const char *pStr, ElemType* pData);
int Struct2String(ElemType *pData, char *pStr);

//主函数
void main()
{
    char szTmp[260] = {0};    //用来接收用户输入的字符串
    ElemType stData;          //字符串转出的结构体用于后面的处理
    for(;;)
    {
        switch(menu_select())
        {
        case 1:
            printf("******************************************\n");
            printf("*         字符串初始化 *\n");
            printf("请输入如:|nnum=xxx|name=xxx|sex=xx|phone=xxx|addr=xxx|
\"字符串,不含\"号\n");
            printf("******************************************\n");
            InitString(szTmp);
            break;
        case 2:
            printf("******************************************\n");
```

```
            printf("*           字符串分析成结构体         *\n");
            printf("****************************************\n");
            printf("*编号(4) 姓名(8) 性别 电话(11) 地址(31)*\n");
            printf("****************************************\n");
            String2Struct(szTmp, &stData);    //字符串到结构体的转换
            printf("%s%s%s%s%s",stData.num,stData.name,
                stData.sex,stData.phone,stData.addr);
            break;
        case 3:
            printf("****************************************\n");
            printf("*           结构体输出为字符串           *\n");
            printf("****************************************\n");

            //结构体赋值
            strcpy(stData.num, "111");
            strcpy(stData.name, "zhangsan");
            strcpy(stData.sex, "2");
            strcpy(stData.phone, "132234523445");
            strcpy(stData.addr, "hangzhou");

            //结构体输出到字符串 szTmp
            Struct2String(&stData, szTmp);
            break;
        case 0:
            printf("\t 再  见! \n");
            return;
        }
    }
}

void FindString( char* pszTmp, const char *pSample, int nlen, char *pDest)
{
    char * pToken = NULL;
    char * p  = NULL;
    pToken = strstr(pszTmp, pSample);
    if (pToken == NULL)
    {
        printf("%s 没有找到\n", pSample);
        return NODATA;
    }

    //查找编号的结束位置
    //pToken + strlen("|num=")为编号的起始地址
```

```
            p = strstr(pToken + strlen(pSample), "|");
            if (p)
            {
                if (p - pToken > strlen(pSample) + nlen)
                {
                    printf("%s 长度超长\n", pSample);
                }
                else
                {
                    strncpy(pDest, pToken + strlen(pSample), p - pToken -strlen
(pSample));     //定长字符串复制,注意没有复制\0,因此前面使用了 memset 函数
                }
            }
        }
        int Struct2String(ElemType *pData, char *pStr)
        {
            char szOut[260] = "字符串为: ";
            _assert(pStr && pData);      //确认 pData, pStr 为

            //拼装字符串
            sprintf(pStr, "|num=%s|name=%s|sex=%s|phone=%s|addr=%s", pData->num,
pData->name, pData->phone, pData->addr);
            strcat(szOut, pStr);          //把字符串 pStr 添加到 szOut 后面
            printf("%s", pStr);
        }
        //字符串解析为结构体
        int String2Struct(const char *pStr, ElemType* pData)
        {
            char * pToken = NULL;
            char * p  = NULL;
            char szTmp[260] = {0};
            char szTmp1[10] = {0};

            if (!pStr||!pData)
            {
                return PTRNULL;
            }
            memset(pData, 0, sizeof(ElemType));       //内存置空

            strcpy(szTmp, pStr);
            strlwr(szTmp);                            //转换成小写字母

            FindString(szTmp, "|num=", sizeof(pData->num), pData->num);
```

```
    FindString(szTmp, "|name=", sizeof(pData->name), pData->name);
    FindString(szTmp, "|sex=", sizeof(pData->sex), pData->sex);
    FindString(szTmp, "|phone=", sizeof(pData->phone), pData->phone);
    FindString(szTmp, "|addr=", sizeof(pData->addr), pData->addr);
}

//接收用户输入的字符串
int  InitString(char *pStr)
{
    char cFlag = 'n';
    char szIn[260] = {0};                          //初始化为全 0

    if (pStr == NULL)                              //参数保护
    {
        printf("不能为空指针！");
        return PTRNULL;
    }

    while ('n' == cFlag )                          //接收用户输入
    {
        scanf("%s", szIn);
        getchar();

        printf("你输入的是：\n\t%s\n", szIn);
        printf("输入正确吗：(y/n)\n");

        scanf("%c", &cFlag);
    }

    strcpy(pStr, szIn);                 //正确赋值返回
    return  SUCESS;
}
```

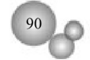



独立实践

1. 请编写一个小程序把下面 URL 中的参数部分(从？字符开始)提取并解析出来。

http://www.ppfilm.cn/player/pfadurl.dll?type=default&uid=12345&pushid=1& partnerid=43256 &fileid=123&filetype=video

2. 请实现系统的 strcpy 函数。



小 结

本章向读者介绍了串类型的定义及其实现方法，并重点讨论了串操作中最常用的“串定位操作(又称模式匹配)”的两个算法。

串有两个显著特点：其一，它的数据元素都是字符，因此它的存储结构和线性表有着很大的不同，如多数情况下，实现串类型采用的是“堆分配”的存储结构，而当用链表存储串值时，结点中数据域的类型不是“字符”，而是“串”，这种块链结构通常只在应用程序中使用；其二，串的基本操作通常以“串的整体”作为操作对象，而不像线性表以“数据元素”作为操作对象。

“串匹配”的简单算法的思想直截了当，简单易懂，适用于在一般的文档编辑中应用，但在某些特殊情况下，如只有 0 和 1 两种字符构成的文本串中应用时效率很低。KMP 算法是它的一种改进方法，其特点是利用匹配过程中已经得到的主串和模式串对应字符之间“等与不等”的信息，以及 *T* 串本身具有的特性来决定之后进行的匹配过程，从而减少了简单算法中进行的“本不必要再进行的”字符的比较。

习 题

一、选择题

1. ______________________称为空串；______________________称为空白串。

2. 设 S=“C:\\document\\datastructure.doc”，则 Strlen(s)=____________，“\”的字符定位的位置为____________。

3. 当且仅当两个串____________________________________，称两串相等。

4. 子串的定位运算称为串的模式匹配；____________称为目标串，____________称为模式。

5. 设目标 T=”abccdcdccbaa”，模式 P=“cdcc”，则第______次匹配成功。

6. 若 n 为主串长，m 为子串长，则串的古典匹配算法最坏的情况下需要比较字符的总次数为______。

二、选择题

1. 下面关于串的叙述中，不正确的是(　　)。
   A. 串是字符的有限序列
   B. 空串即空白串
   C. 模式匹配是串的一种重要运算
   D. 串既可以采用顺序存储，又可以采用链式存储

2. 若串 *S*1="ABCDEFG"，*S*2="9898"，*S*3="###"，*S*4="012345"，执行 concat(replace(S1,substr(S1,length(S2),length(S3)),S3),substr(S4,index(S2,"8"),length(S2)))语句的结果为(　　)。

   A. ABC###G0123　　B. ABCD###2345
   C. ABC###G2345　　D. ABC###2345

E．ABC###G1234　　F．ABCD###1234

G．ABC###01234

3．设有两个串 p 和 q，其中 q 是 p 的子串，求 q 在 p 中首次出现的位置的算法称为(　　)。

A．求子串　　B．连接　　C．模式匹配　　D．求串长

4．若串 S="abbdded"，其子串的数目是(　　)。

A．8　　B．28　　C．29　　D．7

5．串的长度是指(　　)。

A．串中所含不同字母的个数　　B．串中所含字符的个数

C．串中所含不同字符的个数　　D．串中所含非空格字符的个数

6．串"abcabcaaa"的 next 数组为(　　)。

A．−100123410　　B．−101212110　　C．−100012341　　D．−100123012

7．计算上题中字符串的 nextval 为(　　)。

A．(−1,0,0,−1,0,0,−1,4,1)　　B．(−1,0,1,0,1,0,2,1,1)

C．(−1,0,1,0,1,0,0,0,0)　　D．(−1,1,0,1,0,1,0,1,1)

三、简答题

1．空串与空格串有何区别？

2．串是一种特殊的线性表，其特殊体现在什么地方？

3．串的两种最基本的存储方式是什么？

4．两个串相等的充分必要条件是什么？

5．如果两个串含有相等的字符，能否说它们相等？

四、应用题

1．已知：s="(xyz)+*"，t="(x+z)*y"。试利用联结、求子串和置换等基本运算，将 s 转化为 t。

2．两个字符串 $S1$ 和 $S2$ 的长度分别为 m 和 n。求这两个字符串最大共同子串算法的时间复杂度为 $T(m,n)$。估算最优的 $T(m,n)$，并简要说明理由。

3．函数 void insert(char*s,char*t,int pos)将字符串 t 插入到字符串 s 中，插入位置为 pos。请用 C 语言实现该函数。假设分配给字符串 s 的空间足够让字符串 t 插入(说明：不得使用任何库函数)。

4．S="$S_1S_2\cdots S_n$"是一个长为 N 的字符串，存放在一个数组中，编写程序将 S 进行如下修改之后输出。

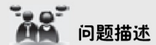
(1) 将 S 的所有第偶数个字符按照其原来的下标从大到小的次序放在 S 的后半部分。

(2) 将 S 的所有第奇数个字符按照其原来的下标从小到大的次序放在 S 的前半部分。

例如，S="ABCDEFGHIJKL"，则修改后的 S="ACEGIKLJHFDB"。



多 维 数 组



问题描述

地雷小游戏

在程序开发中，经常要用到连续的数据，如地雷小游戏中用一个二维数组来记录每一个地雷的数据。

地雷小游戏的规则：根据输入的信息(输入数组的脚标如 3，2)，执行相应的挖雷功能，并根据执行结果给出相应的提示及正确率。

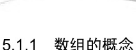
程序界面设计图如图 5.1 所示。



图 5.1 地雷小游戏程序界面设计图

5.1 数　　组

线性表、栈、队列和串都是线性的数据结构，它们具有相似的逻辑特征，即：每个数据元素至多有一个直接前趋和直接后继。而下面要介绍的多维数组是一种复杂的非线性结构。它们的逻辑特征是：一个数据元素可能有多个直接前趋和多个直接后继。



5.1.1 数组的概念

1. 数组的定义

数组(Array)是在程序设计中，为了处理方便，把具有相同类型的若干数据元素 a_0，a_1，…，a_i，…，a_{n-1} 按有序的形式组织起来的一种形式，其中 n 是数组的长度。这些按序排列的同类数据元素的集合称为数组。

2. 数组的分类

根据数组中数据元素 a_i 的组织形式不同，可以将数组分为一维数组、二维数组及多维(n 维)数组。

其中：

(1) 一维数组记为 $A[n]$，由(a_0，a_1，…，a_{n-2}，a_{n-1})n 个具有相同类型的元素组成，每个元素除了值以外还有一个约定元素位置的下标。

(2) 二维数组 $A[m][n]$，由 $m \times n$ 个元素组成，元素之间有规则的排列，每个元素由值和两个能约定元素位置的下标组成，假设 A 是一个有 m 行和 n 列的二维数组，则 A 的表示形式如图 5.2 所示。

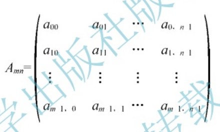


图 5.2 m 行 n 列的二维数组

二维数组可以定义为每个数据元素都是相同类型的一维数组，也可以把二维数组看作由 m 个行向量组成的向量，也可以看作由 n 个列向量组成的向量。其中，每个数据元素 a_j 可以看作一个列向量形式的线性表，

$$a_j = (a_{0j}, a_{1j}, \ldots, a_{m-1j}) \qquad 0 \leqslant j \leqslant n-1$$

或者一个行向量形式的线性表

$$a_i = (a_{i0},\ a_{i1}, \cdots,\ a_{i,n-1}) \qquad 0 \leqslant \mathrm{i} \leqslant m-1$$

由此可见二维数组就是一个具有 m 个或 n 个元素的特殊线性表。并且，每个元素 a_{ij}(除边界外)有两个直接前趋和两个直接后继，即行向量的直接前趋 $a_{i,\ j-1}$ 和直接后继 $a_{i,\ j+1}$，列向量的直接前趋 $a_{i-1,\ j}$ 和直接后继 $a_{i+1,\ j}$。

(3) 多维数组即 n 维数组是“数据元素为 n-1 维数组”的线性表，每个元素均属于 n 个向量，每个元素最多可以有 n 个直接前趋和 n 个直接后继。

5.1.2 数组的存储结构和实现

根据数组的定义可知，数组一旦被定义，其维数和存储空间将不再发生变化，因此，一般不对数组进行插入和删除操作，而一般只对其进行取值和更新操作。

对于数组的基本运算，常见的有以下几种。

(1) 初始化 InitArray(&A，n)：若维数 n 和各维长度合法，则构造相应的数组 A。

(2) 销毁 DestroyArray(&A)：销毁数组。

(3) 取节点 GetElem(A，i)：当数组 A 已存在时，若各个下标不超界，则结果返回 A 中第 i 个数据元素的值，否则返回 FALSE。

(4) 赋值 Assign(&A，i，e)：若下标不超界，则将 e 的值赋给所指定的 A 的元素。

并非任何时候都需要同时执行以上运算。首先，不同问题中的数组所需要执行的运算可能不同；其次，不可能也没有必要给出一组合适各种需要的运算，可以用基本运算的组合来实现。

数组中的元素关系是线性关系，元素之间的位置关系是有规律的，并且数组一般不进行插入或删除操作，所以，采用顺序存储结构存储数组比较合适。但是计算机的内存空间是一维的，因此用一组连续的存储空间来表示多维数组，就必须按某种次序将数组元素排成一个线性序列，然后将这个线性序列顺序存放在内存中。

一维数组的存储方式：假设用一组连续的存储单元存放一个一维数组，并且数组的第一个元素 a_0 的存储地址为 $\mathrm{LOC}(a_0)$，每个元素占用 c 字节，则数组其他元素 a_i 的存储地址 $\mathrm{LOC}(a_i)$ 为

$$\mathrm{LOC}(a_i)=\mathrm{Loc}(a_0)+i\times c$$

多维数组通常有以下两种顺序存储方式。

1. 行优先顺序存储

将数组元素按行向量排列，根据行递增的次序访问数组元素，同一行根据列递增的次序访问数组元素，访问完第 i 行的所有元素之后，再访问第 i+1 行的所有元素。以图 5.2 的二维数组为例，按行优先顺序存储的线性序列为：

$$a_{00}，a_{01}，a_{02}，\cdots，a_{0,n-1}，a_{10}，a_{11}，\cdots，a_{1,n-1}，\cdots\cdots，a_{2n}，\cdots，a_{m-1,0}，a_{m-1,1}，\cdots，a_{m-1,n-1}$$

Java、C 语言中的数组就是按行优先顺序存储的。

2. 列优先顺序存储

将数组元素按列向量排列，根据列递增的次序访问数组元素，同一列根据行递增的次序访问数组元素，访问完第 j 列的所有元素之后，再访问第 j+1 列的所有元素。以图 5.2 的二维数组为例，按列优先顺序存储的线性序列为：

$$a_{00}，\cdots，a_{10}，a_{20}，\cdots，a_{m-1,0}，a_{01}，a_{11}，\cdots，a_{m-1,1}，\cdots\cdots a_{0,n-1}，a_{1,n-1}，\cdots，a_{m-1,n-1}$$

在 FORTRAN 语言中，数组就是按列优先顺序存储的。

根据数组的特性可知，数组的维数及各维的长度被指定后，操作系统会为其分配一片连续的存储空间，并按照上述两种方式中的一种进行存储(具体由设计语言决定)。因此若给出一组下标便可求得相应数组元素的存储位置。

例如，如图 5.2 所示的二维数组按“行优先顺序存储”进行存储，假设每个元素占 L 个存储单元，则数组中的任意元素 a_{ij} 的存储地址计算函数为：

$$\mathrm{LOC}(a_{ij})=\mathrm{LOC}(a_{00})+[n\times i+j]\times L$$

其中，$\mathrm{LOC}(a_{00})$为数组的首地址，也称为基地址或基址。

若数组按“列优先顺序存储”进行存储，则数组中的任意元素 a_{ij} 的存储地址计算函

数为：

$$\mathrm{LOC}(a_{ij})=\mathrm{LOC}(a_{00})+[m\times j+i]$$

根据上述行序优先存储公式，可以推广得到 n 维数组 $A[0\cdots c_1-1, 0\cdots c_2-1, \cdots\cdots, 0\cdots c_n-1]$ 的存储公式，其中 c_i 是第 i 维的长度，因此，对于任意的 $a_{j1j2\cdots jn}$ 的地址计算函数为：

$$\mathrm{LOC}(a_{j1j2\cdots jn})=\mathrm{LOC}(a_{00\cdots0})+(c_2\times\cdots c_n\times j_1+c_3\times\cdots c_n\times j_2+c_n\times j_{n-1}+j_n)L$$

n 维数组的地址计算公式又称为 n 维数组的影像函数。从该函数可以得出，数组元素的存储位置是其下标的线性函数，即给定相应的下标值，计算元素地址的时间复杂度为 $O(1)$，所以数组是一种随机存储结构。

5.1.3 用二维数组解决地雷小游戏的问题

(1) 定义地雷小游戏中数据元素的类型。

```
typedef struct _tagUnit/*定义地雷小游戏中数据元素的类型*/
{
    char bMine;           /*是否是雷*/
    char bOpen;           /*是否已翻开*/
    char nCount;          /*周边有多少雷*/
}ElemType;                /*每块的数据*/
```

(2) 编写程序实现地雷小游戏的各项功能。

```
#include <stdio.h>
#include <time.h>
enum              /*枚举类型*/
{
    NOMINE,       /*0 非雷*/
    ISMINE,       /*雷*/
};
/*函数前置声明*/
int menu_select() ;

int InitMineData(ElemType* pstData);
int PrintMineData(ElemType* pstData);
int PlayMineGame(ElemType* pstData);

/*主函数*/
void main()
{
    char szTmp[260] = {0};           /*用来接收用户输入的字符串*/
    ElemType stData[10][10] ={0};
    int a = 10;
    int b = 10;
```

```
    for(;;)
    {
        switch(menu_select())
        {
        case 1:
            printf("****************************************\n");
            printf("开始初始化地雷区后台数组 *\n");
            InitMineData(stData, a, b); /*初始化雷区*/
            printf("初始化地雷区后台数组完毕\n\n\n");
            break;
        case 2:
            printf("****************************************\n");
            printf("*        地雷分布(1 为雷 0 非雷)\n"              )   ;
            printf("****************************************\n");
            PrintMineData(stData, a, b);   /*输出雷区*/
            break;
        case 3:
            printf("****************************************\n");
            printf("*         玩此游戏(输入数组脚标)       *\n");
            printf("****************************************\n");
            PlayMineGame(stData, a, b); /*玩游戏*/
            break;

        case 4:
            printf("\t 再见!\n");
            return;
        }
    }
}
/**********************************/
/*       菜单选择函数程序          */
/**********************************/
int menu_select()
{
    int sn;
    printf("==============================\n");
    printf("   地雷小游戏\n");
    printf("==============================\n");
    printf("   1.地雷区初始化\n");
    printf("   2.输出地雷分布\n");
    printf("   3.玩此游戏(输入数组脚标如 3,2)\n");
    printf("   4.退出系统\n");
```

```
        printf("==============================\n");
        printf("请选择对应操作的序号：\n");
        for(;;)
        {
            scanf(" %d",&sn);
            if(sn < 0||sn > 4)
                printf("\n\t 输入错误,重选 1~4");
            else
                break;
        }
        return sn;
    }

    int InitMineData(ElemType (*pstData)[10], int a, int b )
    /*初始化雷区*/
    {
        int i = 0;
        int j = 0;
        int ntmp = 0;
        int nCount = 0;
        if (!pstData)
        {
            return -1;
        }

        srand((unsigned)time(NULL));/*以当前时间生成随机数种子,以防每次生成的重复*/
        for (i = 0; i < a; i++)
        {
            for (j = 0; j < b; j++)
            {
                ntmp = rand()%2; /*随机数对 2 求余,0 表示非雷,1 表示雷*/
                pstData[i][j].bMine = ntmp;
                pstData[i][j].bOpen = 0;
            }
        }
        return 0;
    }
    int PrintMineData(ElemType  pstData[10][10], int a, int b)
    /*输出雷区*/
    {
        int i = 0;
        int j = 0;
        int ntmp = 0;
```

```
    int nCount = 0;
    if (!pstData)
    {
        return -1;
    }
    for (i = 0; i < a; i++)
    {
        printf("\t");
        for (j = 0; j < b; j++)
        {
            if (ISMINE == pstData[i][j].bMine)
            {
                printf("%d ", 1);
            }
            else
            {
                printf("%d ", 0);
            }

        }
        printf("\n");
    }

    return 0;
}
int PlayMineGame(ElemType  pstData[10][10], int a, int b)
/*玩游戏*/
{
    int m = 0;
    int n = 0;
    int nMineCount = 0;
    int nNoMineCount = 0;
    char c = 'y';
    int nCount = a*b;/*总块数*/

    while (nMineCount + nNoMineCount < a * b)
    {
        scanf("%d,%d", &m, &n);
        getchar();//
        while((m < 0 || m >= a)|| (n < 0 || n >= b))
        {
            printf("你输入的数超出范围,只能是小于10*10的\n");
            printf("要继续玩吗(y/n)");
            scanf("%c", &c);
            getchar();
```

```
                if (c == 'n')
                {
                    printf("当前正确率是:失败%d/正确%d\n", nMineCount, nNoMineCount);
                    return 0;
                }
                else
                {
                    printf("请输入数组脚标,逗号隔开如 3,2");
                }
                scanf("%d,%d", &m, &n);
                getchar();
            }
            if (!pstData[m][n].bOpen)/*没翻开*/
            {
                pstData[m][n].bOpen = 1;/*表示已翻开*/
                if (pstData[m][n].bMine == ISMINE)/*碰到雷了*/
                {
                    nMineCount ++;
                  printf("你碰到雷了!当前正确率是:失败%d/正确%d\n", nMineCount,
nNoMineCount);
                }
                else/*不是雷*/
                {
                    nNoMineCount++;
                    printf("正确,!当前正确率是:失败%d/正确%d\n", nMineCount,
nNoMineCount);
                }
            }
            else
            {
                printf("此块已翻开过了\n ");
            }
        }


        return 0;
    }
```


独立实践

请把实例中的每一块周边的雷数求出，并赋值到对应块的结构体中。

5.2　矩阵的压缩存储



问题描述

查询城市间距离的问题

对于浙江省的任意六个城市杭州(hz)，温州(wz)，舟山(zs)，台州(tz)，嘉兴(jx)和金华(jh)。如果将这六个城市从1～6进行编号，则任意两个城市之间的距离可以用一个6×6的矩阵来表示。矩阵的第i行和第i列代表第i个城市，distance (i，j)代表城市i和城市j之间的距离。图5.3给出了相应的矩阵。由于对于所有的i和j而言，有distance (i，j)= distance (j，i)，所以该矩阵是一个对称矩阵。

	hz	wz	zs	tz	jx	jh
hz	0	321	229	273	89	183
wz	321	0	354	124	380	232
zs	229	354	0	265	220	325
tz	273	124	265	0	325	216
jx	89	380	220	325	0	242
jh	183	232	325	216	242	0

图5.3　城市间的距离值

要求设计一个简单模拟的查询城市间距离的系统，完成以下功能：输入任意两个城市的名称后，输出城市间的距离值。

在科学与工程计算问题中，矩阵是一种常用的数学对象，通常将矩阵描述为一个二维数组，矩阵在这种存储表示之下，可以对其元素进行随机存取，相应的运算也会非常简单。但是当矩阵中有很多零元素或部分非零元素具有某种分布规律时，称这种矩阵为特殊矩阵，对于特殊矩阵往往要进行压缩存储，以便节省存储空间。所谓压缩存储即多个值相同的元素只分配一个存储空间，不存储零元素。特殊矩阵主要有对称矩阵、三角矩阵、对角矩阵等。

5.2.1　特殊矩阵的逻辑结构

1. 对称矩阵

假如A是一个n阶矩阵，若其元素满足如下性质：

$$0\leqslant i \quad j\leqslant n-1$$

则称A为n阶对称矩阵。图5.4便是一个4阶对称矩阵。

$$\begin{bmatrix} 2 & 4 & 6 & 0 \\ 4 & 1 & 9 & 5 \\ 6 & 9 & 4 & 7 \\ 0 & 5 & 7 & 0 \end{bmatrix}$$

图5.4　4阶对称矩阵

对称矩阵中的元素关于主对角线对称，故只需要为每一对对称的元素分配一个存储空间，对于 n 阶矩阵中的 n^2 个元素只需要 $n(n+1)/2$ 个存储单元，这样能节约近一半的存储空间。对于 n 阶矩阵，我们按“行优先顺序存储”其主对角线(包括对角线)以下的元素，其存放形式如图 5.5 所示。

0	1	2	3	…	$\frac{n(n-1)}{2}-1$	…	$\frac{n(n+1)}{2}-1$
a_{00}	a_{10}	a_{11}	a_{20}	…	$a_{n-1,0}$	…	$a_{n-1,n-1}$

图 5.5　n 阶对称矩阵的压缩存储

在这个下三角矩阵中，第 i 行 $(0\leqslant i\leqslant n-1)$ 恰有 $i+1$ 个元素，元素总数为：

$$\sum_{i=0}^{n-1}(i+1)=\frac{n(n+1)}{2}$$

以此类推，假设将 n 阶对称矩阵存放在一个一维数组 $\text{sa}\left[\frac{n(n+1)}{2}\right]$ 中，那么矩阵中的任意元素 a_{ij} 和数组元素 sa[k]之间的对应关系为：

$$\text{LOC}(a_{ij})=\begin{cases}\text{LOC}(a_{00})+\dfrac{i\times(i+1)}{2}+j & 0\leqslant j\leqslant i\leqslant n-1\\ \text{LOC}(a_{00})+\dfrac{j\times(j+1)}{2}+i & 0\leqslant i\leqslant j\leqslant n-1\end{cases}$$

2．三角矩阵

以主对角线划分，三角矩阵有上三角和下三角两种。上三角矩阵是指其对角线以下的元素为常数 c 或零的 n 阶矩阵。下三角矩阵与之相反，是指其对角线以上的元素为常数 c 或零的 n 阶矩阵，如图 5.6 所示。

$$\boldsymbol{A}_{n\times n}=\begin{bmatrix}a_{0,0} & a_{0,1} & \cdots & a_{0,n-1}\\ c & a_{1,1} & \cdots & a_{1,n-1}\\ \vdots & \vdots & \vdots & \vdots\\ c & c & c & a_{n-1,n-1}\end{bmatrix}\qquad \boldsymbol{B}_{n\times n}=\begin{bmatrix}a_{0,0} & c & \cdots & c\\ a_{0,1} & a_{1,1} & \cdots & c\\ \vdots & \vdots & \vdots & \vdots\\ a_{0,n-1} & a_{1,n-1} & \cdots & a_{n-1,n-1}\end{bmatrix}$$

(a) 上三角矩阵　　　　(b) 下三角矩阵

图 5.6　三角矩阵

三角矩阵的压缩存储与对称矩阵基本一样，只是除了存储其上三角或下三角的元素之外，还要加一个存储常数 c 的存储空间。因此，三角矩阵可压缩存储到向量 $\text{sa}\left[\frac{n\times(n+1)}{2}+1\right]$ 中，c 存放在向量的最后一个存储单元中。

因此，sa[k]和 a_{ij} 的对应关系式如下。

(1) 上三角矩阵的 sa[k]和 a_{ij} 的对应关系：

$$\text{LOC}(a_{ij})=\text{LOC}(a_{00})+\frac{i\times(i+1)}{2}+j\qquad 0\leqslant j\leqslant i\leqslant n-1$$

(2) 下三角矩阵的 sa[k]和 a_{ij} 的对应关系是：

$$\mathrm{LOC}(a_{ij})=\mathrm{LOC}(a_{00})+\frac{j\times(j+1)}{2}+i \qquad 0\leqslant i\leqslant j\leqslant n-1$$

3. 对角矩阵

对角矩阵是指所有的非零元素都集中在以主对角线为中心的带状区域中，即除了主对角线上和主对角线相邻两侧的若干条对角线上元素之外，所有其他的元素均为零。常见的对角矩阵有三对角矩阵、五对角矩阵、七对角矩阵等。图 5.7 就是一个三对角矩阵。

$$\begin{bmatrix} a_{00} & a_{01} & & \cdots & \\ a_{10} & a_{11} & a_{12} & \cdots & 0 \\ & a_{21} & a_{22} & \cdots & \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & & \cdots & a_{n-1,n-2} \end{bmatrix}$$

图 5.7 三对角矩阵

5.2.2 用特殊矩阵解决查询城市间距离的问题

该运算可用如下形式算法描述。

```
#include "stdafx.h"
#include "stdlib.h"
#include "string.h"
#include "stdio.h"

int main()
{
    /*初始化城市名称数组*/
    char cityname[6][3]={{'h','z','\0'},{'w','z','\0'},{'z','s','\0'},
{'t','z','\0'},{'j','x','\0'},{'j','h','\0'}};
    int i=0,j=0;
    int distance[15];
    printf("请输入城市矩阵的下三角矩阵:");
    for (int m = 0; m < 15; m++)
    {
        scanf("%d",&distance[m]);
    }

    char* from = (char*)malloc(sizeof(char)*3);
    char* to = (char*)malloc(sizeof(char)*3);
    char flag=' ';
    do
    {
        printf("请输入源城市的简写名称:");
        scanf("%s",from);
        for (int m = 0; m < 6;m++ )
        {
            if (strcmp(cityname[m],from)==0)
```

```
                {
                    i = m;
                    break;
                }
            }

            printf("请输入目的城市的简写名称:");
            scanf("%s",to);
            for (int m = 0; m < 6; m++)
            {
                if (strcmp(cityname[m],to)==0)
                {
                    j = m;
                    break;
                }
            }

            printf("%s 到%s 的距离是:%d\n",cityname[i],cityname[j],i>j?
                distance[i*(i-1)/2+j]:i==j?0:distance[j*(j-1)/2+i]);
/*计算城市间距离*/
            printf("还要继续查询吗(y/n)?");
            getchar();
            flag=getchar();
        }while (flag == 'y' || flag == 'Y');
        return 0;
}
```

独立实践

查找离给定城市距离最近的城市。

5.3 稀疏矩阵

5.3.1 稀疏矩阵的逻辑结构

稀疏矩阵是指矩阵($A_{m\times n}$)中非零元素的个数(s)远远小于矩阵中元素的总数($s=m\times n$)，且非零元素分布无规律。$\delta=\dfrac{s}{m\times n}$ 为矩阵的稀疏因子，通常 $\delta\leqslant 0.05$ 的矩阵称为稀疏矩阵。图 5.8 即为一个稀疏矩阵。

$$\boldsymbol{A}_{4\times 5}=\begin{bmatrix}5&0&0&0&3\\0&0&0&2&0\\0&0&7&0&0\\4&0&0&0&0\end{bmatrix}$$

图 5.8 稀疏矩阵

5.3.2 稀疏矩阵的压缩存储

根据稀疏矩阵的定义可知，稀疏矩阵的压缩存储会失去随机存取功能。因此，对于稀疏矩阵有两种常用的存储方法：三元组表和十字链表。

1. 三元组表

1) 三元组

在存储稀疏矩阵时，为了节省存储单元，压缩存储方式可只存储非零元素。但由于非零元素的分布一般是没有规律的，因此，在存储非零元素的同时，还必须存储适当的辅助信息，才能迅速确定这个非零元素是矩阵中的哪一个元素。最简单的方法是将非零元素的值和它所在的行号、列号作为一个节点存放在一起，于是矩阵中的每一个非零元素都由一个三元组(i，j，a_{ij})唯一确定。其结构如下。

行号	列号	元素值
row	column	value

因此，图 5.8 所示的稀疏矩阵可以用如下三元组进行描述。

((0，0，5)，(0，4，3)，(1，3，2)，(2，2，7)，(3，0，4))

显然，要唯一确定一个稀疏矩阵，还必须存储该矩阵的行数和列数。为了运算方便，还要将非零元素的个数与三元组表存储在一起。因此，有如下的类型说明。

```
#define MAXSIZE 100                    /*非零元素的个数*/
typedef struct
{
    int i,j;                           /*非零元素的行号、列号*/
    DataType e;                        /*非零元素值*/
}TripleNode;
typedef struct
{
    int m,n,r;                    /*稀疏矩阵的行数、列数及非零元素的个数*/
    TripleNode data[MAXSIZE+1];   /*三元组顺序表*/
}TMatrix
```


三元组线性表可以采用顺序存储结构或链式存储结构进行存储。

2) 三元组顺序表

若将稀疏矩阵中的非零元素按行优先(或列优先)的顺序存放在一个由三元组组成的数组中进行存储，则得到一个节点均是三元组的线性表，称该表为三元组顺序表。该表是稀疏矩阵的一种顺序存储结构。图 5.6 所示的稀疏矩阵的三元组表实现如图 5.9 所示。

2. 十字链表法

当矩阵中非零元素的位置或个数经常发生变化(如矩阵进行加减法等运算)时，用三元组顺序表来存储稀疏矩阵中的非零元素就不太合适了。此时，采用链表做存储结构更为恰当。

稀疏矩阵的链表存储方法主要有三种：单链表、行/列的单链表和十字链表。下面仅介绍十字链表的表示方法。

i	j	e
0	0	5
0	4	3
1	3	2
2	2	7
3	0	4

图 5.9 稀疏矩阵 A 的三元组存储

在该方法中，每一个非零元素对应十字链表中的一个节点。每个节点的结构如图 5.10 所示。

行	列	值	行后继结点	列后继结点
i	j	e	rnext	cnext

图 5.10 稀疏矩阵的十字链表节点结构

其中，i、j、e 分别指非零元素 e 在稀疏矩阵中所对应的行号为 i，列号为 j。rnext 指示本行的下一个非零元素，cnext 指示本列的下一个非零元素。

行指针域 rnext 将稀疏矩阵中同一行上的非零元素链接在一起，列指针域 cnext 将同一列上的非零元素链接在一起。因此元素 a_{ij} 既属于第 i 行也属于第 j 列，是第 i 行和第 j 列的交叉点，故称这样的链表为十字链表。为了运算方便，增加了两个指针数组，分别用来存放行的单链表的头指针和列的单链表的头指针。图 5.8 中矩阵 A 的十字链表如图 5.11 所示。

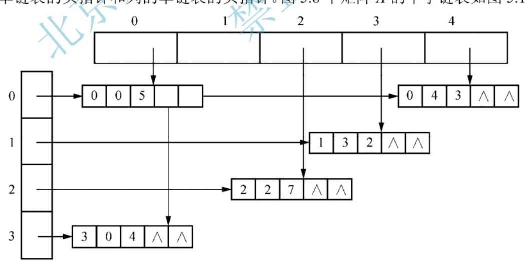


图 5.11 图 5.8 的稀疏矩阵 A 的十字链表

十字链表的说明如下：

```
typedef struct LNode
{
    int i,j;
```

```
        DataType e;
        struct LNode *rnext,*cnext;
    }Link;
    Link *rhead [4],*chead[5];
```

小　　结

通过本章的学习，读者熟悉了多维数组和特殊矩阵的类型定义及其在C语言中的实现方法。数组作为一种数据类型，是一种多维的非线性结构，并只进行存取或修改某个元素的值，因此一般顺序存储在一组地址连续的存储单元中，下标和存储地址有一定的对应关系。研究非零元素或零元素分布有一定规律的特殊矩阵的压缩存储对于解决我们现实生活的问题有比较重要的意义，如可用于图像压缩方面。

习　　题

一、填空题

1．一维数组的逻辑结构是________，存储结构是________；对于二维或多维数组，分别按________和________两种不同的存储方式进行存储。

2．对于一个二维数组 $A[m][n]$，若按行序为主序存储，每个元素占 k 个存储单元，并且第一个元素的存储地址是 LOC($A[0][0]$)，则 $A[i][j]$的地址是________。

3．二维数组 $A[10][20]$采用列序为主方式存储，每个元素占 1 个存储单元，并且第一个元素的存储地址是 200，则 $A[6][12]$的地址是________。

4．有一个 10 阶整型对称矩阵 $\boldsymbol{A}$，采用压缩存储方式(以行序为主序存储，且 $A[0][0]=1$)，则 $A[8][5]$的地址是________。

5．设矩阵 $\boldsymbol{A}$ 是一个对称矩阵，为了节约存储空间，将其下三角矩阵按行序存放在一维数组 $B[0，n(n+1)/2]$中，对下三角部分中的元素 $a_{ij}(i>j)$而言，在一维数组 B 中下标 k 的值是________。

6．三元组表中的每个节点对应于稀疏矩阵的一个非零元素，它包含三个数据项，分别表示该元素的________、________和________。

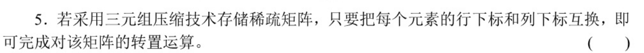
二、判断题

1．一维数组和线性表的区别是前者长度固定，后者长度可变。　(　　)

2．在二维数组中，每个数组元素同时处于一个向量中。　(　　)

3．多维数组实际上是由一维数组实现的。　(　　)

4．多维数组的数组元素之间的关系是线性的。　(　　)



5．若采用三元组压缩技术存储稀疏矩阵，只要把每个元素的行下标和列下标互换，即可完成对该矩阵的转置运算。　　(　　)

三、选择题

1．常对数组进行的两种基本操作是(　　)。

A．索引与修改　　B．新建与删除

C．存取和修改数据元素　　D．查找与索引

2．在二维数组 *A* 中，每个元素的长度为 3 个字节，行下标为 0～7，列下标为 0～9，从首地址 sa 开始连续存放在存储器内，该数组按行存放时，数组元素 a[7][4]的起始地址为(　　)。

A．1240　　B．1222　　C．1140　　D．1320

3．设有一个 10 阶的对称矩阵 ***A***，采用压缩存储方式，以行序为主存储，$a_{1,1}$ 为第一个元素，其存储地址为 1，每个元素占一个地址空间，则 $a_{8,5}$ 的地址为(　　)。

A．33　　B．21　　C．15　　D．52

4．一个 $n \times n$ 的对称矩阵，如果以行或列为主序放入内存，则容量为(　　)。

A．$n \times n$　　B．$n \times n/2$　　C．$n \times (n+1)/2$　　D．$(n+1)^2/2$

5．二维数组 *M* 的成员是六个字符(每个字符占一个字节)组成的串，行下标 *i* 为 0～8，列下标 j 为 1～10，则存放 *M* 至少需要(①)个字节；*M* 的第 8 列和第 5 列共占(②)个字节；若 M 按行优先方式存储，则元素 *M*[8][5]的起始地址与当 *M* 按列优先方式存储时的(③)元素的起始地址一致。

① A．540　　B．520　　C．80　　D．90

② A．240　　B．108　　C．60　　D．100

③ A．*M*[8][5]　　B．*M*[7][5]　　C．*M*[8][4]　　D．*M*[3][10]

6．稀疏矩阵一般的压缩存储方法有(　　)。

A．二维数组和三维数组　　B．三元组和散列

C．散列和十字链表　　D．三元组和十字链表

四、简答题

1．按行优先顺序和列优先顺序列出四维数组 *A*[2][2][2][2]所有元素在内存中的存储次序。

2．现有稀疏矩阵 ***A*** 如下所示，要求画出其以下各种表示法。

$$\begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 3 & 5 & 0 \\ 0 & 0 & 0 & 7 \\ 6 & 0 & 0 & 0 \end{bmatrix}$$

(1) 三元组顺序表表示法。

(2) 十字链表表示法。

五、编程题

1．以一维数组作为存储结构，实现线性表的就地逆置，即在原表的存储空间内将线性表(a_1，a_2，a_3，…，a_{n-1}，a_n)逆置为(a_n，a_{n-1}，…，a_3，a_2，a_1)。

2．已知一个 $n \times n$ 的矩阵 $\boldsymbol{B}$ 按行优先存于一个一维数组 $A[0 \cdots n(n-1)]$中，试给出一个就地转置算法，使其将原矩阵转置后仍存于数组 $\boldsymbol{A}$ 中。

3．假设稀疏矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$(具有相同的大小 $m \times n$)都采用三元组表示，编写一个函数计算 $\boldsymbol{C}=\boldsymbol{A}+\boldsymbol{B}$，要求 $\boldsymbol{C}$ 也采用三元组表示。





树

问题描述

快速搜索磁盘文件中记录的问题

有一个文件包含很多条记录，如图 6.1(a)所示，每条记录都有一个唯一的关键字(学号)标识该条记录。该文件随机存储在磁盘上，为了方便实现在磁盘上对图 6.1(a)中的记录进行增、删、改、查操作，一般需要建立一张与之对应的索引表(见图 6.1(b))。

现需要实现如下的功能。

(1) 选择一种数据结构在内存中存放索引表，通过该数据结构能高效地插入、遍历和搜索索引表；(2)输入任一关键字(学号)，显示该关键字的物理记录地址。

物理记录地址	学号	姓名	班级	其他
1000	20120101	王一涛	计算机 121	……
1050	20120102	张巧红	计算机 121	……
1100	20120103	邵建军	计算机 121	……
1150	20120104	刘依美	计算机 121	……
1200	20120105	陈佳	计算机 121	……
1250	20120106	刘悠悠	计算机 121	……
1300	20120107	王亦非	计算机 121	……

(a) 文件存储区

	关键字	物理记录地址
1	20120101	1000
	20120103	1100
	20120105	1200
2	20120102	1050
	20120104	1150
	20120106	1250
	20120107	1300

(b) 索引表

图 6.1 随机存储中的记录

6.1 概 述

树形结构是一类重要的非线性结构。树形结构是节点之间有分支，并具有层次关系的结构，它非常类似于自然界中的树。树结构在客观世界中是大量存在的，如家谱、行政组织机构都可用树形象地表示。树在计算机领域中也有着广泛的应用，如在编译程序中，用树来表示源程序语法结构；在数据库系统中，可用树来组织信息。本章重点讨论二叉树的存储表示及其各种运算，并研究一般树、森林与二叉树的转换关系，最后介绍树的应用实例。



1. 树的定义

1) 树的定义

树(Tree)是 $n(n>0)$个节点的有限集合 T，$n=0$ 的树称为空树，当 $n>0$ 的时候满足如下两个条件：

(1) 有且仅有一个称为根(Root)的节点。

(2) 其余的节点可分为 $m(m\geqslant 0)$个互不相交的有限集合 T_1，T_2，…，T_m，其中每个集合又是一棵树，并称其为根节点的子树(Sub Tree)。

在树的树形图表示中，节点通常是用圆圈表示的，节点的名称一般是写在圆圈旁边的，有时也可写在圆圈内，如图 6.2(a)所示。

树的定义是一个递归的定义，即一棵树是由若干棵子树构成的，而子树又可由若干棵更小的子树构成。树中的每个节点都是该树中某一棵子树的根。

常见的树的表示方法有图示法、集合表示法和横向凹入表示法，如图 6.2 所示。

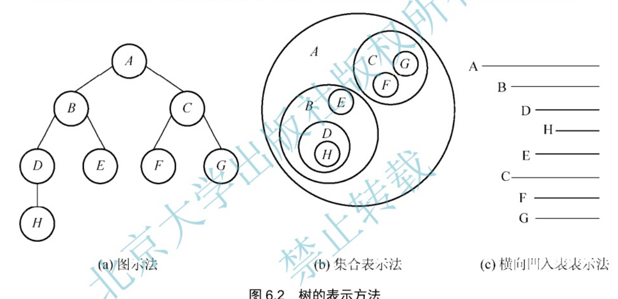


图 6.2 树的表示方法

从图 6.2 可以看出，树的表示方法中图示法最直观，该方法主要用来描述树的逻辑结构，横向凹入法主要用于树的屏幕显示和打印输出，本书主要采用图示法来表示树的结构。

2) 树的特点

非空的树具有下面的特点。

(1) 有且仅有一个节点没有直接前趋，称它为根节点。

(2) 有一个或多个节点没有直接后继，称它为终端节点，即叶子节点；

(3) 除开始节点外，树中其他任一节点都有且仅有一个直接前趋；

(4) 除终端节点外，树中其他任一节点都有一个或多个直接后继。

2. 树的基本术语

下面以图 6.2(a)为例给出树结构中常用的基本术语。

(1) 节点：树中包含的一个数据元素及若干指向其子树的分支，如图 6.2(a)中的 A、B、C 等。

(2) 节点的度：一个节点所具有的子树的个数，如图 6.2(a)中的 *A* 节点，其度为 2。

(3) 树的度：树中度数最大的节点的度，如图 6.2(a)所示的树，其度为 2。

(4) 叶子：度为 0 的节点即为叶子节点(又称终端节点)，如图 6.2(a)中的 *E*、*F*、*G*、*H*。

(5) 非终端节点：也称内部节点，是指除根节点外，其他度不为 0 的节点，如图 6.2(a)中的 *B*、*C*、*D*。

(6) 双亲、孩子节点：树中某个节点的子树的根称为该节点的孩子，相应地，该节点称为孩子的双亲。在图 6.2(a)中，*B* 的双亲为 *A*，*B* 的孩子为 *D*、*E*。

(7) 兄弟节点：同一个双亲的孩子之间互称为兄弟，如图 6.2(a)中的 *B*、*C*。

(8) 祖先：从根到该节点所经分支上的所有节点，如图 6.2(a)中 *H* 的祖先为 *A*、*B*、*D*。

(9) 子孙：以某一节点为根的子树中的任意节点都称为该节点的子孙，如图 6.2(a)中 *B* 的子孙为 *D*、*E*、*H*。

(10) 节点的层数：从根开始算起的层数。根节点的层数为 1，其余节点的层数等于其双亲节点的层数加 1。在图 6.2(a)中，*A* 的层数为 1，*B*、*C* 的层数为 2，*D*、*E*、*F*、*G* 的层数为 3，*H* 的层数为 4。

(11) 树的高度：树中节点的最大层数，如图 6.2(a)中树的高度为 4。

(12) 无序树、有序树：如果将树中各个节点的子树看作从左到右有顺序的(即不能互换的)，则称该树为有序树；否则称为无序树。在以后的章节里，若不特别指明，我们所讨论的树都是有序树。

(13) 森林：$m(m \geqslant 0)$棵互不相交的树的集合。如图 6.2(a)所示，去掉 *A* 节点就可以得到两棵树(分别以 *B*、*C* 为根节点)构成的森林，因此，对于树中的每个节点而言，其子树的集合即为森林，即删去一棵树的根，就得到一个森林。反之，加上一个节点作为树根，森林就变成一棵树。

6.2 二 叉 树

二叉树是一个非常重要的树形结构，一般的树也能简单地转换为二叉树，由于二叉树的存储结构及其算法都比较简单，因此经常被用来解决实际中的树形问题，是一种常用的树形结构。

6.2.1 二叉树的定义

1. 二叉树的定义

二叉树(Binary Tree)是 $n(n \geqslant 0)$个节点所构成的有限集合，当 $n=0$ 时称为空二叉树；当 $n>0$ 时由一个根节点及两棵互不相交的、分别称为左子树和右子树的子二叉树组成。

2. 二叉树的特点

二叉树中的子树具有左右之分，并且每个节点最多只能有两棵子树。所以二叉树的度最多为 2。并且在二叉树中，即使是一个孩子也有左右之分。

3. 二叉树的五种基本形态

根据二叉树的定义，可以得出二叉树具以下有五种基本形态，如图 6.3 所示。

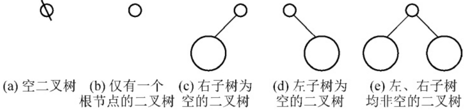


图 6.3 二叉树的五种基本形态

6.2.2 二叉树的性质

下面给出二叉树的几个重要性质。

性质 1 在二叉树第 i 层上最多有 2^{i-1} ($i\geqslant1$)个节点。

证明：(数学归纳法)。

(1) 当 i=1 时，只有一个根节点，并且 $2^{i-1}=2^{1-1}=1$，所以命题成立。

(2) 假设对所有的 j($1\leqslant j<i$)命题成立，即第 j 层上至多有 2^{j-1} 个节点，那么可以证明 $j=i$ 时命题也成立。

(3) 根据归纳假设，第 i−1 层上至多有 2^{i-2} 个节点。由于二叉树的每个节点至多有两棵子树，故第 i 层上的节点数，最多是第 i−1 层上的最大节点数的 2 倍，即 $j=i$ 时，该层上的最大节点数为 $2\times 2^{i-2}=2^{i-1}$，因此命题成立。

性质 2 深度为 k 的二叉树至多有 2^k-1 个节点($k\geqslant1$)。

证明：由性质 1 可知，深度为 k 的二叉树的节点数最多为：

$$\sum_{i=1}^{k}2^{i-1}=2^k-1$$

性质 3 对任意一棵二叉树，如果其叶子节点的个数为 n_0，度为 2 的节点的个数为 n_2，则 $n_0=n_2+1$。

证明：假设二叉树中节点总数为 n，度为 1 的节点数为 n_1，则有：

$$n=n_0+n_1+n_2 \tag{6.1}$$

又因为在二叉树中度为 1 的节点有一个孩子，度为 2 的节点有两个孩子，但树中只有根节点不是任何节点的孩子，并且叶子节点没有孩子节点，故二叉树中的节点总数又可表示为：

$$n=0\times n_0+1\times n_1+2\times n_2+1 \tag{6.2}$$

综述所述可得：$n_0=n_2+1$。

下面介绍两种特殊情形的二叉树：满二叉树和完全二叉树。

满二叉树(Full Binary tree)：深度为 k 且有 2^k-1 个节点的二叉树称为满二叉树。图 6.4 是一个深度为 4 的满二叉树。

满二叉树的特点是每一层上的节点数都达到最大值，即对给定的高度 k，它的节点数都为最大值 2^k-1。满二叉树中的每个分支节点均有两棵高度相同的子树，且树叶都在最下一层上。

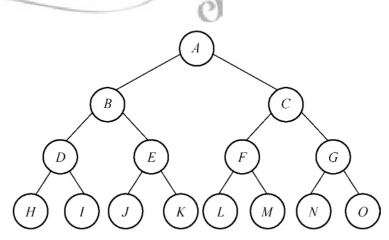


图 6.4　满二叉树

完全二叉树(Complete Binary Tree)是深度为 k 的，有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中的编号从 $1\sim n$ 一一对应时，称该树为完全二叉树。图 6.5 是一个深度为 4 的完全二叉树。

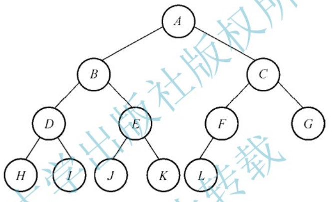


图 6.5　完全二叉树

显然满二叉树是完全二叉树，但完全二叉树不一定是满二叉树。

深度为 k 的完全二叉树的特点如下：

(1) 叶子节点只可能出现在第 k 层或 $k-1$ 层上。

(2) 对任一节点，若其右分支下的子孙的最大层次为 h，则其左分支下的子孙的最大层次为 h 或 $h+1$。

因此，在完全二叉树中，若某个节点没有左孩子，则它一定没有右孩子，即该节点必是叶子节点。如图 6.6 所示，节点 F 没有左孩子而有右孩子 M，所以它不是一棵完全二叉树。

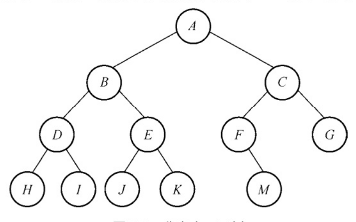


图 6.6　非完全二叉树

性质 4 具有 n 个节点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。(符号 $\lfloor x \rfloor$ 表示不大于 x 的最大整数)

证明：设所求完全二叉树的深度为 k，根据其定义和性质 2 所知：

$$2^{k-1}-1 < n \leqslant 2^k - 1$$

由此可推出：$2^{k-1} \leqslant n < 2^k$。对不等式两边取对数后有

$$k-1 \leqslant \log_2 n < k$$

因为 k 是整数，故有：$k = \lfloor \log_2 n \rfloor + 1$。

性质 5 一棵具有 n 个节点的完全二叉树(其深度为 $\lfloor \log_2 n \rfloor + 1$)，如果按照从上往下、从左至右的顺序对二叉树中的所有节点从 1 开始编号，则对于任意节点 $i(1 \leqslant i \leqslant n)$ 有如下特点。

(1) 若 i=1，则 i 为二叉树的根节点，无双亲；若 i>1，则 i 的双亲节点为 $\lfloor i/2 \rfloor$。

(2) 若 $2i>n$，则节点 i 无左孩子(节点 i 为叶子节点)，否则 i 的左孩子节点为 $2i$。

(3) 若 $2i+1>n$，则节点 i 无右孩子，否则 i 的右孩子节点为 $2i+1$。

例如，在图 6.5 中，当 i=1 时为根节点 A，其左孩子节点 B 为 $2i$=2，右孩子节点 C 为 $2i+1$=3。

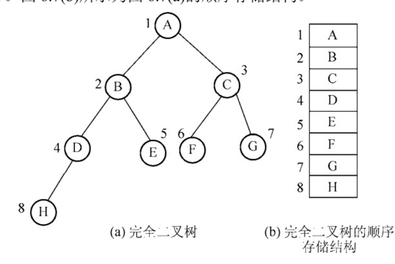
6.2.3　二叉树的存储结构

类似于线性表，二叉树也可以采用顺序和链式两种存储方式。

1. *顺序存储结构*

这种存储方式较适用于完全二叉树和满二叉树。顺序存储是将二叉树的所有节点，按照一定的节点顺序，存储到一个连续的存储单元中。因此，为了能反映出各个节点之间的逻辑关系，必须把节点安排成一个适当的线性序列。

对于具有 n 个节点的完全二叉树，从根节点开始自上而下，从左到右，给所有节点从 1 开始编号，就可以得到一个反映整个二叉树结构的线性序列，如图 6.7(a)所示。在完全二叉树中对于任意节点的编号 i，根据二叉树的性质 5 可推得其双亲节点、左右孩子节点、兄弟等节点的编号。图 6.7(b)所示为图 6.7(a)的顺序存储结构。

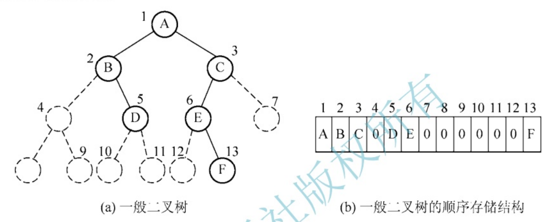


(a) 完全二叉树　　(b) 完全二叉树的顺序存储结构

图 6.7　节点编号的完全二叉树及其顺序存储结构

由二叉树的性质可知，完全二叉树和满二叉树中节点的层次序列完全反映了节点之间的逻辑关系。而顺序存储结构依靠数组中的序号位置准确地反映了节点之间的逻辑关系。因此顺序存储结构能够存储完全二叉树。

显然，利用顺序存储结构存储完全二叉树，既简单又节省存储空间。但是，对于一般的二叉树采用顺序存储时，为了能够明确地表示树中各个节点之间的逻辑关系，也必须按照完全二叉树的结构将各个节点存储在一维数组的相应分量中，图 6.8 所示为非完全二叉树及其顺序存储结构。



(a) 一般二叉树　　(b) 一般二叉树的顺序存储结构

图 6.8　一般二叉树及其顺序存储结构

图 6.8(b)中的“0”表示不存在此节点，顺序存储方式对于非完全二叉树的存储造成了存储空间的浪费。因为，最坏的情况是一个深度为 k 的且只有 k 个节点的右单支树却需要 2^k-1 个节点的存储空间。因此这种存储方式仅适用于完全二叉树，而非完全二叉树更适合使用链式存储结构。

2. 链式存储结构

二叉树通常采用链式存储结构存储二叉树中的节点及其相互之间的关系，根据二叉树的定义可知，链表中的每个节点除了存储数据元素外，还至少要有两个指针域分别指向其左右孩子节点，才能表示二叉树的层次关系，比较常用的链式存储结构有二叉链表、三叉链表。图 6.9(a)所示为二叉链表中的节点结构。在此结构中对于任意节点而言，寻找其左右孩子节点非常方便，但是寻找其双亲节点又比较困难，需要从根指针出发逐个排查。所以，在具体应用中，如果经常要在二叉树中寻找某节点的双亲，则可以在每个节点上再加一个指向其双亲的指针域 parent，图 6.9(b)所示即为三叉链表中的节点结构。

lchild	date	rchild

(a) 二叉链表的结点结构

lchild	date	parent	rchild

(b) 三叉链表的结点结构

图 6.9　二叉树的节点及其存储结构

利用以上两种节点结构所得到的二叉树的存储结构即为二叉链表和三叉链表。

二叉链表是二叉树的最常用的存储结构，在后面的各小节中的有关二叉树的各种算法，

都是基于这种存储结构的。下面给出二叉链表中节点结构在C语言中的类型定义。

```
typedef struct BiTNode
{
    DataType data;                        //节点的数据元素
    struct BiTNode *lchild,*rchild;       //左右孩子指针
} BiTNode, *BiTree;
```

二叉树的二叉链表中的所有节点类型为BiTNode，链表的头指针root指向二叉树的根节点，图6.10(a)、(b)所示为二叉树及其二叉链表存储结构。若二叉树为空，则root=NULL。若节点的某个孩子不存在，则相应的指针域为空。容易证得，在含有 n 个节点的二叉树中，一共有 $2n$ 个指针域，其中只有 $n-1$ 个用来指示节点的左、右孩子，其余的 $n+1$ 个指针域为空，那么利用这些空指针域存储其他有用信息，可以得到另外一种称之为线索链表的存储结构。

树形结构的存储方法有好几种，至于哪种方式合适，主要根据实际问题实际操作进行选择。例如，如果需要频繁地查找节点的双亲，则需要采用三叉链表存储，如图6.10(c)所示为图6.10(a)的三叉链表存储结构。

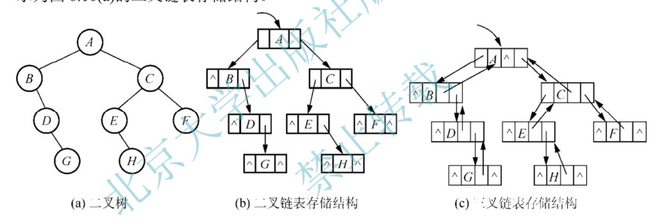


图6.10　二叉树及其二叉链表、三叉链表存储结构

6.3　二叉树的遍历和线索化

6.3.1　二叉树的遍历

1. 遍历二叉树的定义及算法描述

二叉树的遍历是指沿某条搜索路径访问二叉树中的每个节点，并且对树中每个节点仅访问一次。对节点的访问可以是输出、更新、增加、删除等操作。由前面章节可知，对于一个线性结构的遍历很容易，只需要从开始节点出发顺序扫描每个节点即可。由于二叉树的每个节点可以有两个孩子节点，因此，需要寻找一种规律来访问树中各节点。

根据二叉树的定义可知，一棵非空的二叉树由三部分组成：根节点、左子树和右子树。若分别用D、L和R表示上述三部分，则对于一棵非空二叉树的遍历即为：访问根节点，

遍历左子树，遍历右子树，则有 DLR、LDR、LRD、DRL、RDL、RLD 六种次序的遍历方案。其中前三种遍历序列与后三种遍历序列正好相反，前三种按先左后右的次序遍历根的两棵子树，后三种则按先右后左的次序遍历根的两棵子树，由于二者对称，在算法设计上没有本质区别，因此只讨论前三种次序的遍历方案。

根据前三种遍历方案中根节点访问的顺序，可得到二叉树的三种遍历次序：先根遍历(或先序遍历)、中根遍历(或中序遍历)及后根遍历(或后序遍历)。

根据二叉树的递归定义，可得二叉树的三种遍历的递归算法。

1) 先根遍历二叉树

若二叉树为空则遍历操作结束，否则依次进行如下操作：访问根节点，先根遍历左子树，先根遍历右子树。

2) 中根遍历二叉树

若二叉树为空则遍历操作结束，否则依次进行如下操作：中根遍历左子树，访问根节点，中根遍历右子树。

3) 后根遍历二叉树

若二叉树为空则遍历操作结束，否则依次进行如下操作：后根遍历左子树，后根遍历右子树，访问根节点；

例 6.1 已知表达式 a+b*c-f/(d-e)对应的遍历二叉树如图 6.11 所示，请写出其对应的三种遍历次序。实现该运算的基本思想：根据二叉树的递归定义及遍历序列中根节点的访问次序可得：先根遍历序列为−+a*bc/f−de。中根遍历序列为 a+b*c−f/d−e。后根遍历序列为 abc*+fde−/−。

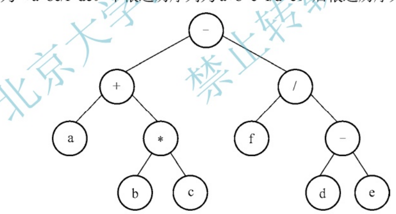


图 6.11　表达式的二叉树

在下面的三个递归算法中，二叉树为空作为递归的终止条件，此时应为空操作。根节点的访问操作应当根据具体的情况而定，在此不妨假设访问根节点时输出节点数据。若以二叉链表作为存储结构，则遍历算法如下。

算法 6.1　先根遍历的递归算法。

```
void PreOrderTraverse(BiTree T)
{
 /*先根遍历二叉树*/
    if(T)
```

```
        {
            printf("%c",T->data);
            PreOrderTraverse(T->lchild);
            PreOrderTraverse(T->rchild);
        }
    }
```

算法 6.2 中根遍历的递归算法。

```
    void InOrderTraverse(BiTree T)
    {
     /*中根遍历二叉树*/
        if(T)
        {
        InOrderTraverse(T->lchild);
        printf("%c",T->data);
        InOrderTraverse(T->rchild);
        }
    }
```

算法 6.3 后根遍历的递归算法。

```
    void PostOrderTraverse(BiTree T)
    {
     /*后根遍历二叉树*/
        if(T)
        {
          PostOrderTraverse(T->lchild);
          PostOrderTraverse(T->rchild);
          printf("%c",T->data);
        }
    }
```


除了上述 3 种遍历方式以外，二叉树还有一种层次遍历方式。该方式是以“从上到下，从左到右”的次序对二叉树进行遍历的，即竖直方向上按照树的高度(根节点作为第一层)一层层向下进行访问，同一层上按照从左到右的方向进行访问。如图 6.11 中的二叉树，其对应的层次遍历序列为−+/a*f−bcde。

二叉树的遍历除了递归方法外，还有非递归的算法，非递归算法执行效率较高，并且能清晰地看出遍历的执行过程，下面以二叉树的中根遍历为例，给出其非递归的遍历算法。根据遍历二叉树的特点可知，用栈结构可以很容易地实现非递归遍历算法。

算法 6.4 中根遍历的非递归算法。

```
    Void InorderTraverse(BiTree T){  //中序遍历二叉树
    InitStack(&S);p=T;
    While(p||!StackEmpty(S)){
        If(p){                            //若根节点非空,则根节点入栈,遍历其左子树
```

```
            Push(S,p);
            P=p->lchild;
            }
      Else{                                   //若根节点为空,则访问根节点,遍历其右子树
         Pop(S,p);
         printf("%c",p->data);
         p=p->rchild;
         }
      }
   }
```

2. 根据遍历序列构造二叉树

(1) 二叉树的先根序列和中根序列可以唯一的确定一棵二叉树。

根据二叉树的遍历序列定义可知，在对二叉树进行先根遍历后得到的序列中，第一个节点肯定是二叉树的根节点(Root)。在其对应的中根序列中找到 Root 结点，该节点可以将中根序列划分成前后两个子序列，Root 节点之前的部分是左子树的中根序列(LeftPart)，之后的部分是右子树的中根序列(RightPart)。

再根据这两个子序列找到先根序列中对应的两部分。在先根序列中的左子序列的第一个节点肯定是左子树的根节点(LeftRoot)，右子序列的第一个节点肯定是右子树的根节点(RightRoot)，然后分别在两个子序列中找到对应的位置，两个根结点(LeftRoot 和 RightRoot)又可将 LeftPart 及 RightPart 分别分成前后两部分。如此继续下去，直到所有子序列划分为空，便可得到一棵唯一的二叉树。

(2) 二叉树的中根序列和后根序列可以唯一的确定一棵二叉树。

同理，根据二叉树的遍历序列定义可知，在对二叉树进行后根遍历后得到的序列中，最后一个节点肯定是二叉树的根节点。在其对应的中根序列中找到 Root 节点，该节点可以将中根序列划分成前后两个子序列，Root 节点之前的部分是左子树的中根序列，之后的部分是右子树的中根序列。

再根据这两个子序列找到后根序列中对应的两部分。在后根序列中的左子序列的最后一个节点肯定是左子树的根节点，右子序列的最后一个节点肯定是右子树的根节点，然后分别在两个子序列中找到对应的位置，两个根结点又可将 LeftPart 及 RightPart 分别分成前后两部分。如此继续下去，直到所有子序列划分为空，便可得到一棵唯一的二叉树。

注意：已知二叉树的先根序列和后根序列，无法唯一确定一棵二叉树。因为如果无法确定左右子树，也就无法确定节点之间的层次关系及左右次序关系，也就不能确定这棵二叉树，空树和只有一个节点的二叉树除外。

例 6.2 已知一棵二叉树的先根序列和中根序列分别为 *abcdefgh* 和 *cbafegdh*，请画出这棵二叉树。

实现该运算的基本思想：根据上面的定义可知，先根序列中的第一个节点为二叉树的根节点，即二叉树的根节点为 *a*，利用 *a* 将中根序列划分成前后两个子序列，即 *cb* 和 *fegdh*，

再根据这两个子序列找到后根序列中对应的两部分 *bc* 和 *defgh*。先根序列中的左子序列的第一个节点肯定是左子树的根节点，即 *b* 节点为左子树的根节点，右子序列的第一个节点肯定是右子树的根节点，即 *d*，然后分别在两个子序列中找到对应的位置，两个根节点(*b* 和 *d*)又可将两个子序列(*cb* 和 *fegdh*)分别划分成前后两部分，如此继续下去，直到所有子序列划分为空，便可得到一棵唯一的二叉树，如图 6.12 所示。

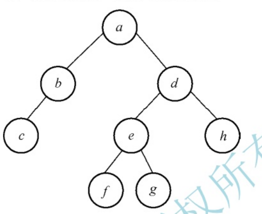


图 6.12　由先根序列和后根序列确定的一棵二叉树

如果已知二叉树的中根遍历序列为 *ecbhfdjiga*，后根遍历序列为 *echfjigdba*，能确定一棵唯一的二叉树吗？如果可以请给出构造方法。

建议读者思考，如果已知二叉树的先根序列为 *abc*，后根序列为 *cba*，请问能确定一棵唯一的二叉树吗？请给出理由。

6.3.2　二叉树的线索化

二叉树的遍历其实是将二叉树中的节点按照一定的线性序列进行排列，在该序列中能够找到每个节点(除第一个和最后一个节点外)的直接前趋和直接后继。但是当二叉树采用二叉链表作为其存储结构时，因为每个节点中只有指向其左、右孩子节点的指针域，只能直接找到该节点的左、右孩子信息，而一般情况下无法直接找到该节点在任一序列中的前趋和后继节点，因此这种信息只有在遍历的动态过程中才能得到。因此，如果要查找某个节点在任一序列中的前趋或后继节点时可以采用以下两种方式。

(1) 重新遍历二叉树。

(2) 在每个节点中增加两个指针域来存放遍历时得到的前趋和后继节点信息。

以上两种方式要么消耗时间，使得效率降低；要么浪费存储空间。那么解决上述问题的好方法是什么呢？答案就是线索二叉树。

1. 线索二叉树的定义

由 6.2.3 小节可知，在 *n* 个节点的二叉链表中含有 *n*+1 个空指针域，可以利用这些空指针域，存放节点在某种遍历次序下的前趋和后继的信息，这种指向其前趋和后继的指针域称为“线索”，加上线索的二叉树称为线索二叉树(Threaded Binary Tree)。

2. 线索二叉树的节点定义

为了区分一个节点的指针域到底是指向其孩子的还是指向线索的，需要在每个节点中增加两个线索标志域，节点的结构由以下五个部分组成。

lchild	ltag	data	rtag	rchild

其中：

$$ltag=\begin{cases}0 & \text{lchild域指示节点的左孩子}\\1 & \text{lchild域指示节点的直接前趋}\end{cases}$$

$$rtag=\begin{cases}0 & \text{rchild域指示节点的右孩子}\\1 & \text{rchild域指示节点的直接后继}\end{cases}$$

在如图 6.13(a)所示的中根线索二叉树中，它的线索链表如图 6.13(b)所示。图中的实线表示指针，虚线表示线索。节点 *C* 的左线索为空，表示 *C* 是中根序列的开始节点，它没有前趋；节点 *E* 的右线索为空，表示 *E* 是中根序列的终端节点，它没有后继。显然，在线索二叉树中，一个节点是叶子节点的充要条件为：它的左、右线索标志均是 1。

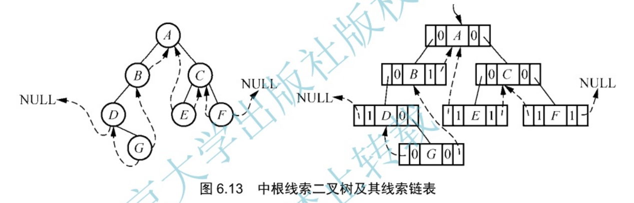


图 6.13 中根线索二叉树及其线索链表

下面给出线索链表中节点结构在 C 语言中的类型定义。

```
    typedef  int  DataType;   /* DataType 是线索二叉树节点中数据的类型,可以为整型、
字符型等类型,此处假设为整型*/
    typedef struct BiThrNode
    {
        DataType data;                      //结点的数据元素
        struct BiThrNode *lchild,*rchild;   //左右孩子指针
        int ltag,rtag;
    } BiThrNode, *BiThrtree;
```

由该节点结构构成的二叉链表作为二叉树的存储结构时，称之为线索链表。

3. 二叉树的线索化

对二叉树按照一定的次序进行遍历，对节点的操作是检查当前节点的左右指针域是否为空，若为空则用线索取代空指针，使其指针域指向其直接前趋或后继节点，并修改其线索标识域为 1，该过程称为二叉树的线索化。

为此，增加一个指针 pre，使其始终指向刚刚访问过的节点，指针 *p* 指向当前正在访问的节点，则有节点*pre 是节点**p* 的前趋，而**p* 是*pre 的后继。

下面以中根线索二叉树为例给出线索化的算法：线索化的过程其实就是遍历的过程，唯一区别在于线索化时对节点的操作是修改节点的空指针使其指向前趋或后继的线索。

算法 6.5　二叉树的中根线索化。

算法描述如下。

若节点*p 的指针域为空，则修改其 ltag(或 rtag)为 1。

若节点*p 的前趋节点*pre!=NULL，则进行如下操作。

① 若节点*pre 的右线索标志已建立(即*pre->rtag==1)，则令 pre->rchild 指向其中序前趋节点*p 的右线索。

② 若节点*p 的左线索标志已建立(即*pre->ltag==1)，则令 p->lchild 指向其中序前趋节点*pre 的左线索。

③ 将 pre 指向刚刚访问过的节点*p(即 pre=p)，则在下一次访问一个新节点*p 时，*pre 为其前趋节点。

根据线索链表中的节点结构，下面给出中根线索化算法在 C 语言中的定义。

```
BiThrNode *pre;                                  /*全程量,初值应为 NULL*/
void InThreading(BiThrNode *p)
{
    if(p)
    {
    InThreading(p->lchild);                      /*递归左子树线索化*/
        if (p->lchild==NULL) p->ltag=1;          /*建立左线索标志*/
        if (p->rchild==NULL) p->rtag=1;          /*建立右线索标志*/
        if(pre! =NULL)
          {
            if(pre->rtag==1)                     /* *p 无右子树*/
            pre->rchild=p;                       /*右线索 pre->rchild 指向*p*/
            if(p->ltag==1)                       /**p 无左子树*/
            p->lchild=pre;                       /*左线索 p->rchild 指向*pre*/
        }
    pre=p;
    InThreading(p->rchild);                      /*递归右子树线索化*/
    }
}                                                /*二叉树的中序线索化算法*/
```

读者可以根据此算法写出先根线索化和后根线索化算法，请课下思考。

4. 线索二叉树的遍历

在线索二叉树中直接查找某节点的前趋和后继节点时不必重新遍历，使得查找效率大大提高了。可以很方便地求得节点在先根、中根和后根次序下的前趋节点和后继节点。

(1) 求节点 p 在中根线索二叉树中的后继节点。

① 若 p 的右子树为空，即 p->rtag=1，则 p->rchild 指向 p 的后继节点。

② 若 p 的右子树不为空，即 p->rtag=0，根据中根遍历的规律可知，p 的后继必是其右

子树中第一个中序遍历的节点，即右子树中最左下的节点。如图 6.13 所示，*A* 的后继节点是 *E*。

算法 6.6 在中根线索二叉树中求节点直接后继的算法。

```
BiThrNode *InorderNext(BiThrNode *p)     /*求中根线索二叉树中 p 的后继节点*/
                                         /*函数返回指向中序后继的指针*/
{
if(p->rtag==1)                  /*p 的右子树为空*/
    p=p->rchild;                /p->rchild 是右线索,指向 p 的后继*/
else                            /* p 的右子树非空*/
{
    p=p->rchild;                /*从 p 的右孩子开始查找*/
    while(p->ltag==0)           /*当 p 不是左下节点时,继续查找*/
    p=p->lchild;
}
    return p;
}  /*InorderNext*/
```

(2) 求节点 *p* 在中根线索二叉树中的前趋节点。

① 如果 *p* 的左子树为空，即 p->ltag=1，则 p->lchild 指向 *p* 的前趋节点。

② 如果 *p* 的左子树不为空，即 p->ltag=0，根据中根遍历的规律可知，*p* 的前趋节点必是其左子树中第一个中序遍历的结点，即子树中最右下的节点。如图 6.13 所示，*A* 的中序前趋节点是 *B*。

算法 6.7 在中根线索二叉树中求节点直接前趋的算法。

```
BiThrNode *InorderPre(BiThrNode *p)  /*求中根线索二叉树中 p 的前趋节点*/
                                 /*函数返回指向中根前趋的指针*/
{
if(p->ltag==1)                   /*p 的左子树为空*/
    p=p->lchild;  /p->lchild 是左线索,指向 p 的前趋*/
else                             /* p 的左子树非空*/
{
    p=p->lchild;                 /*从 p 的左孩子开始查找*/
    while(p->rtag==0)            /*当 p 不是右下节点时,继续查找*/
    p=p->rchild;
}
return p;
}  /*InorderPre*/
```

(3) 求节点 *p* 在后根线索二叉树中的前趋节点。

① 如果 *p* 的左子树为空，即 p->ltag=1，则 p->lchild 指向 *p* 的前趋节点。在图 6.14 中，*H* 的前趋是 *B*，*F* 的前趋是 *G*。

② 如果 *p* 的左子树不为空，即 p->ltag=0，则：

若 p 的右子树不为空，则 p->rchild 所指节点(即其右孩子节点)为其前趋，在图 6.14 中，A 的前趋是 E；

若 p 的右子树为空，则 p->lchild 所指节点(即其右孩子节点)为其前趋，在图 6.14 中，E 的前趋是 F。

(4) 求节点 p 在后根线索二叉树中的后继节点。

① 如果 p 是根节点，则其后继为空。

② 如果 p 是其双亲的右孩子，则 p 的双亲节点为其后继节点，在图 6.14 中，E 的后继是 A。

③ 如果 p 是其双亲的左孩子，而且 p 没有兄弟节点，则 p 的双亲节点为其后继节点，在图 6.14 中，F 的后继是 E。

④ 如果 p 是其双亲的左孩子，但是 p 有兄弟节点，则 p 的后继是其双亲的右子树中后根遍历的第一个节点，即右子树中“最左下的叶子节点”，如图 6.14 中，B 的后续后继是双亲 A 的右子树中最左下的叶子节点 H，注意 F 是该子树中“最左下”的节点，但它不是叶子。

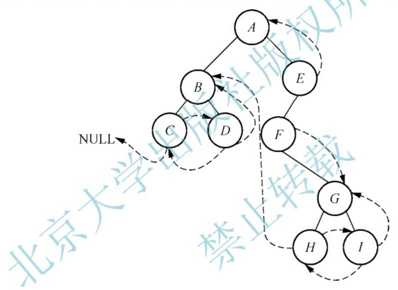


图 6.14 后根线索二叉树

由上述讨论可知，在后根线索树中，仅从 p 出发就能找到其后根前趋节点；而查找 p 的后根后继节点，仅当 p 的右子树为空时，才能直接由 p 的右线索 p->rchild 得到，否则必须知道 p 的双亲节点才能找到其后续后继节点。因此，如果线索二叉树中的节点没有指向其双亲节点的指针，就可能要从根节点开始进行后根遍历才能找到节点 p 的后续后继。由此可见，线索对查找指定节点的后续后继并无多大帮助。

思考题：写出后根线索二叉树中求节点前趋和后继的算法。

(5) 求节点 p 在先根线索二叉树中的前趋节点。

① 如果 p 的左子树为空，即 p->ltag=1，则 p->lchild 指向 p 的前趋节点。

② 如果 p 的左子树不为空，即 p->ltag=0，则：如果 p 是双亲的左孩子，那么 p 的前趋是其双亲节点；如果 p 是双亲的右孩子，那么 p 的前趋是其双亲左子树中先根遍历的最后一个节点。

(6) 求节点 p 在先根线索二叉树中的后继节点。

① 如果 p 的右子树为空，即 p->rtag=1，则 p->rchild 指向 p 的后继节点。

② 如果 *p* 的右子树不为空，即 p->rtag=0，则：如果 *p* 有左子树，那么 *p* 的后继是其左子树的根；如果 *p* 没有左子树，那么 *p* 的后继是其右子树的根。

算法 6.8 写出遍历中根线索二叉树的算法。

算法描述：遍历某种次序的线索二叉树，只要从该次序下的开始节点出发，反复找到节点在该次序下的后继，直至终端节点。这对于中根和先根线索二叉树而言是十分简单的，无需像非线索树的遍历那样，引入栈来保存留待以后访问的子树信息。

```
void TraverseInorderThread(BiThrNode *p)     /*遍历中根线索二叉树 p*/
{
   if(p!=NULL)                               /*非空树*/
     {while(p->ltag==0)                      /*找中根序列的开始结点*/
      p=p->lchild;
     do
      {printf("\t%d\n",p->data);             /*访问结点*p*/
       p=InorderNext(p);                     /*找*p 的中序后继结点*/
       }while(p!=NULL);
   }
}/* TraverseInorderThread */
```

由于中根序列的终端节点的线索为空，所以 do 语句终止条件是 p 为 NULL，显然该算法的时间复杂性为 $O(n)$，但常数因子比上节讨论的遍历算法小，且无需设栈，因此，若要对一棵二叉树经常遍历或查找节点在指定次序下的前趋和后继，其存储结构采用线索树为宜。

本节介绍的线索二叉树都是既有左线索又有右线索的，但在许多应用中常常只用到右线索，而没有必要设立左线索。

6.3.3 用二叉树解决快速搜索磁盘文件中记录的问题

(1) 定义数据元素类型。

```
typedef struct
{
    long stuId;                          /*学号*/
    long memoryAdd;                      /*内存地址*/
} Record, *RecordType;                   /*二叉树内存记录节点数据及其指针类型*/

typedef struct BiTNode
{
    RecordType data;                     /*节点的数据元素*/
    struct BiTNode *lchild, *rchild; /*左右孩子指针*/
} BiTNode, *BiTree;
```

(2) 编写程序实现各项功能。

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

typedef struct
{
    long stuId;                     /*学号*/
    long memoryAdd;                 /*内存地址*/
} Record, *RecordType;              /*二叉树内存记录节点数据及其指针类型*/

typedef struct BiTNode
{
    RecordType data;                /*节点的数据元素*/
    struct BiTNode *lchild, *rchild;/*左右孩子指针*/
} BiTNode, *BiTree;

/*创建二叉树*/
BiTree CreateBiTree()
{
    BiTree root = NULL;
    long stuId = 0;
    long memoryAdd = 0;

    scanf("%ld", &stuId);
    scanf("%ld", &memoryAdd);
    if(stuId != 0 && memoryAdd != 0)
    {
        RecordType data = (RecordType)malloc(sizeof(Record));
        data->stuId = stuId;
        data->memoryAdd = memoryAdd;

        root = (BiTree)malloc(sizeof(BiTNode));
        root->data = data;
        root->lchild = CreateBiTree();
        root->rchild = CreateBiTree();
    }

    return root;
}

/*先根遍历二叉树*/
void PreOrderTraverse(BiTree t)
{
```

```
            if(t != NULL)
            {
                printf("学号:%ld, 内存地址:%ld\n", t->data->stuId, t->data->
memoryAdd);
                PreOrderTraverse(t->lchild);
                PreOrderTraverse(t->rchild);
            }
        }

        /*根据学号查找树节点*/
        BiTNode* FindNode(BiTree t, long stuId)
        {
            BiTNode* result = NULL;
            if(t != NULL)
            {
                if(t->data->stuId == stuId)
                {
                    result = t;
                }
                else
                {
                    result = FindNode(t->lchild, stuId);
                    if(result == NULL)
                    {
                        result = FindNode(t->rchild, stuId);
                    }
                }
            }

            return result;
        }


        int main()
        {
            char ch = '\0';
            BiTree tree = NULL;
            do
            {
                printf("********请选择所需要的操作*********\n");
                printf("1. 构建二叉树\n");
                printf("2. 先根遍历二叉树\n");
                printf("3. 根据学号查找内存地址\n");
```

```
            printf("4. 退出\n");
            printf("\n");

            ch = getchar();
            getchar();
            switch (ch)
            {
            case '1':
                {
                    printf("请按照先根顺序输入节点;如果没有子节点,则输入 0\n");
                    tree = CreateBiTree();
                    break;
                }
            case '2':
                {
                    PreOrderTraverse(tree);
                    break;
                }
            case '3':
                {
                    printf("请输入学号:");
                    long stuId = 0;
                    scanf("%ld", &stuId);
                    if(stuId > 0)
                    {
                        BiTNode *result = FindNode(tree, stuId);
                        if(result != NULL)
                        {
                            printf("学号%ld 对应记录的内存地址为%ld\n\n",
stuId, result->data->memoryAdd);
                        }
                        else
                        {
                            printf("不存在该学号\n\n");
                        }
                    }
                    else
                    {
                        printf("学号必须大于零\n\n");
                    }
                    break;
                }
            case '4':
```

```
            {
                exit(1);
                break;
            }
            default:
                exit(1);
            }

            flushall();

        }   while (true);

        return 0;
    }
```



独立实践

试给出二叉树中根、后根遍历的算法。

6.4 树 和 森 林

现实中的问题往往只能用树和森林进行描述，而不能直接用二叉树进行表示，因此本节将介绍二叉树与森林之间的对应关系，并给出树的存储表示及其遍历。

6.4.1 树的存储

树的存储形式有多种，本节仅讨论树的常用的三种表示法，即双亲链表表示法、孩子链表表示法及孩子兄弟链表表示法。

1. 双亲链表表示法

双亲链表表示法是指利用树中每个节点都具有唯一的双亲节点的性质，用一个连续的存储空间来存储树中的节点信息，同时为每个节点附加一个指针域 parent，用于指向其双亲节点所在的位置，这样就可唯一地表示一棵树。其节点的结构如图 6.15 所示。

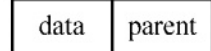


图 6.15 双亲链表表示法中节点的结构

下面给出双亲链表表示法中节点结构，在 C 语言中的类型定义。

```
#define MAX_TREE_NODE  100        /*定义树中节点数目的最大值*/
typedef struct PTnode
{
 elemtype data;                   /*数据域*/
```




```
    int parent;                    /*双亲位置域*/
}PTnode;
Typedef struct PTree
{
    PTnode nodes[MAX_TREE_NODE];
    int n;                         /*树中的节点数*/
}
```

图 6.16 所示为一棵树及其双亲链表表示的结点结构。

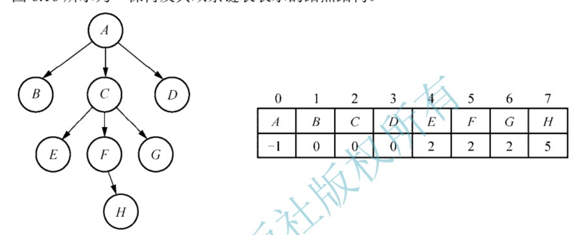


图 6.16 树及其双亲链表表示法

由图 6.16 可知，双亲链表表示法中的指针 parent 是向上链接的，因此查找某个节点的双亲节点变得非常容易，但是要查找其孩子节点就需要重新遍历整个结构。下面的孩子链表表示法可以方便查找孩子节点。

2. 孩子链表表示法

由于树中每个节点可以有多个孩子，因此，当采用多重链表来表示度为 k 的树时，每个节点内要设置 k 条链指向其孩子节点。在 n 个节点的树中，其空指针域的数目是 $kn-(n-1)=n(k-1)+1$，这将造成极大的空间浪费。图 6.16 中度为 3 的树的存储结构如 6.17 所示，其对应的空指针域为 17。

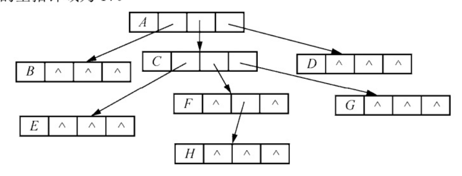
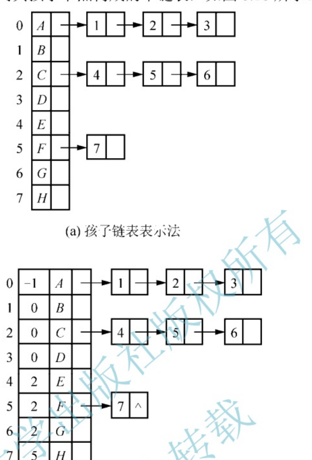


图 6.17 树的孩子链表表示法

如果根据每个节点实际的孩子数设置指针域，并在节点内增加一个度数域用来表示该节点所指示的节点数，虽然节省了储存空间，但是给运算带来了不便。

那么最好的表示方法是将每个节点的孩子节点进行排列，将其看成一个线性表，且以

单链表作为其存储结构，则 n 个节点有 n 个孩子链表(叶子的孩子链表为空表)，同时在每个节点上附加一个指针指向其孩子节点构成的单链表，如图 6.18 所示。



(a) 孩子链表表示法

(b) 孩子双亲链表表示法

图 6.18　图 6.16 中树的孩子链表表示法及孩子双亲链表表示法

下面给出孩子链表表示法中节点的结构在 C 语言中的类型定义。

```
typedef struct CTNode         //孩子节点结构
 {
        int child;            /*孩子结点序号*/
   struct CTNode *next;
  }*ChildPtr;                 /*孩子链表结点*/
typedef struct                //双亲结点结构
 {
        DataType data;        /*树结点数据*/
     ChildPtr firstchild;     /*孩子链表头指针*/
  }CTBox;
typedef struct
{
        CTBox  nodes[MAX_TREE_NODE];
}
```

```
        int n;
    }
```

孩子链表表示法对于查找某个节点的孩子非常容易，但是查找其双亲节点又比较麻烦，为此可以把双亲链表表示法和孩子链表表示法结合起来，即孩子双亲链表表示法，该方法在双亲节点结构中附加一个双亲指示域，如图 6.18(b)所示。

3. 孩子兄弟链表表示法

孩子兄弟链表表示法和二叉树的二叉链表表示完全一样，它采用两条链分别连接第一个孩子和其下一个兄弟节点，分别命名为 firstchild 和 rightsibling，即可得树的孩子兄弟链表表示结构。其节点结构如图 6.19 所示。

第一个孩子结点	数据域	右兄弟结点
firstchild	data	rightsibling

图 6.19　孩子兄弟链表表示法中节点的结构

下面给出孩子兄弟链表表示法中节点结构在 C 语言中的类型定义。

```
    typedef struct CSnode
    {
     Datatype data;  /*数据域*/
     Struct CSnode *firstchild,*rightsibling;  /*用来指示节点的第一个左孩子和下
一个兄弟节点*/
    }CSnode,*CStree;
```

例如，图 6.16 中树的孩子兄弟链表如图 6.20 所示。这种存储结构的最大优点是，它可以方便地实现树和二叉树的相互转换及树的各种操作，因此，可利用二叉树的算法来实现对树的操作。

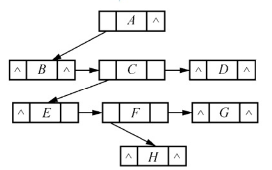


图 6.20　图 6.16 中树的二叉链表表示法

例如，若要访问节点的第 i 个孩子节点，只需要沿着 firstchild 指针找到第一个孩子节点，然后沿着孩子节点的 rightsibling 指针连续走 i-1 步即可。这种存储结构实际上是将一棵树转换成一棵对应的二叉树，这样有利于实现树和森林的各种操作。因此这是一种较为普遍的树的存储方式。

6.4.2 树、森林与二叉树的转换

从树的孩子兄弟链表表示的定义可知，这种存储结构其实就是将一棵树转换成一棵二叉树，然后进行存储。因此，一棵树能够唯一地转换成一棵二叉树；反之，一棵二叉树也能够还原成唯一的一棵树。此外，因为树的根节点没有兄弟节点，因此在树转换得到的二叉树中，其根节点的右子树必为空。如果将森林中的每一棵树的根节点看作同一层的有序排列的兄弟节点，则森林也可以转换成一棵二叉树。

1. 树、森林转换成二叉树

树中每个结点可能有多个孩子，但二叉树中每个结点最多只能有两个孩子。要把树转换为二叉树，就必须找到一种结点与结点之间至多用两个量说明的关系。树中每个结点最多只有一个最左边的孩子(长子)和一个右邻的兄弟，这就是我们要找的关系。按照这种关系很自然地就能将树转化成对应的二叉树：

(1) 在所有兄弟结点之间加一连线；

(2) 对每个结点，除了保留与其长子的连线外，去掉该结点与其它孩子的连线。

使用上述变换法，图 6.21(a)所示的树就变为图(b)的形式，它已是一棵二叉树，若按顺时针方向将它旋转约 45° 就更清楚地变为图(c)所示的二叉树。由于树根没有兄弟，故树转化为二叉树后，二叉树的根结点的右子树必为空。

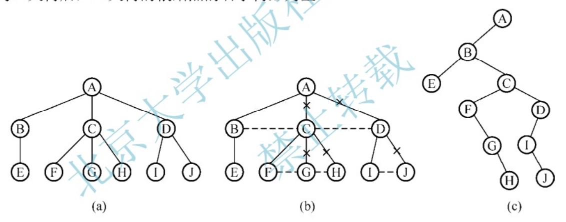


图 6.21 树转化成二叉树

将一个森林转换为二叉树的方法是：现将森林中每一棵树变为二叉树，然后将各二叉树的根结点视为兄弟连在一起。例如在图 6.22 中，(b)是(c)的转换结果，(c)是(b)旋转后的二叉树。

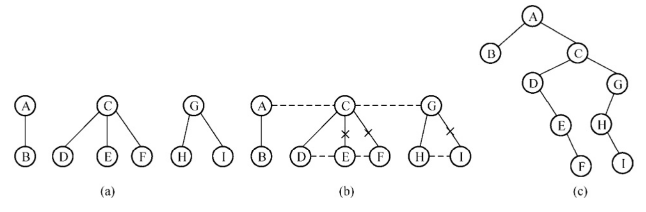


图 6.22 森林转化成二叉树

2. 二叉树到树、森林的转换

同样，也有一种自然的方式把二叉树转换到树和森林；若结点 x 是其双亲 y 的左孩子，则把 x 的右孩子，右孩子的右孩子，……，都与 y 用连线连起来，最后去掉所有双亲到右孩子的连线。如图 6.23 所示，将(a)所示二叉树通过上述方法转换成(c)所示森林。

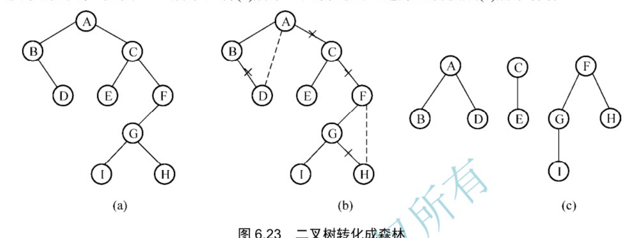


图 6.23 二叉树转化成森林

6.4.3 树和森林的遍历

树和森林主要有两种遍历方式：先根遍历和后根遍历。

1. 树的遍历

(1) 先根遍历过程。
① 访问树的根节点。
② 按照从左到右的顺序先根遍历根节点的各子树。
(2) 后根遍历过程。
① 按照从左到右的顺序后根遍历根节点的各子树。
② 访问树的根节点。

例如，对图 6.21(a)所示的树进行前根遍历和后根遍历，得到的序列分别是 ABCEFGHDIJ 和 BEHFGCIJDA。

请读者思考，该树对应的二叉树的先根、中根和后根遍历序列与该树的先根、后根遍历序列之间有何关系。

2. 森林的遍历

(1) 先根遍历森林过程如下。
若森林非空，则：
① 访问森林中第一棵树的根节点。
② 先根遍历第一棵树的根节点的子树森林。
③ 先根遍历除第一棵树之外其他的树构成的森林。
(2) 后根遍历森林过程如下。
① 后根遍历第一棵树的根节点的子树森林。
② 访问森林中第一棵树的根节点。
③ 后根遍历除第一棵树之外其他的树构成的森林。

例如，对图 6.22(a)所示的森林进行先根遍历和后根遍历，得到的序列分别是 ABCDEFGHI

和 BADEFCHIG。

请读者思考，该森林对应的二叉树的先根、中根和后根遍历序列与该森林的先根、后根遍历序列之间有何关系。

6.5　哈夫曼树及其应用



问题描述

文件传输编码问题

现实生活中为了文件的安全及传输速度，常需要对文件进行加密和压缩，哈夫曼(Huffman)编码是数据压缩技术中的一种无损压缩方法，它既实现了加密又具有压缩功能。常用于文本、图像的压缩，如 TXT 文件、JPG 文件等。

现需要实现如下的功能：利用哈夫曼编码实现文件传输时的编码及解码问题。

本节以哈夫曼(Huffman)树为例介绍二叉树的具体应用。哈夫曼树又称为最优二叉树，是一种应用非常广泛的树结构。

6.5.1　基本概念

哈夫曼树是一类带权路径长度最短的树，具有广泛的应用。其所涉及的基本概念如下。

(1) 路径：指树中的一个节点到另一个节点之间的分支序列。

(2) 路径长度：指路径上的分支数目。

(3) 树的路径长度：从根节点到树中每个节点的路径长度之和。

(4) 节点的权：在实际应用中为表达某种实际意义，而为节点赋予的一个实数。

(5) 节点的带权路径长度：指该节点到根节点之间的路径长度乘以该节点上的权值。

(6) 树的带权路径长度：树中所有叶子节点的带权路径长度之和，通常记为：

$$\mathrm{WPL}=\sum_{i}^{n} w_i \times l_i$$

其中，n 表示树中叶子节点的总个数，W_i 和 L_i 分别表示第 i 个叶子节点的权值和根到该叶子节点的路径长度。n 个带权叶子节点所构成的二叉树中，WPL 最小的二叉树称为最优二叉树或哈夫曼树。

例 6.3 以权值分别为 6、7、3、2 的四个节点作为叶子节点，可以构造多棵编码的二叉树，且带权路径长度不同，如图 6.24 所示。

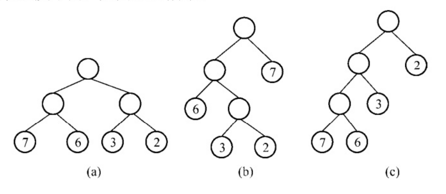


图 6.24　带权路径长度不同的二叉树

(a) WPL=7×2+6×2+3×2+2×2=36。

(b) WPL=7×1+6×2+3×3+2×3=34。

(c) WPL=7×3+6×3+3×2+2×1=47。

其中，(b)树的 WPL 最小，它就是哈夫曼树。可以验证，将权值大的节点尽量靠近根节点，会使得 WPL 的值最小。

6.5.2 哈夫曼树的构造

1. 哈夫曼树的构造方法

哈夫曼树的构造过程可以描述如下。

(1) 由给定的 n 个权值 W_1，W_2，W_3，…，W_n 构成 n 棵二叉树的森林 F={T_1，T_2，T_3，…，T_n}，其中每棵二叉树 T_i 中都只有一个权值为 W_i 的根节点。其左右子树均空。

(2) 重复以下步骤，直到 F 中只剩下一棵二叉树为止，该树便是哈夫曼树。

① 在 F 中选出两棵根节点的权值最小的二叉树(当这样的树不止两棵树时，可以从中任选两棵)分别作为左右孩子来构造一棵新的二叉树，新二叉树的根节点的权值为左右孩子权值之和。

② 在 F 中删除这两颗二叉树，并将新的二叉树加入到 F 中。

例 6.4 四个叶子节点的权值分别为 6、7、3、2，根据上述方法构造一棵哈夫曼树，给出其构造过程。

构造过程如图 6.25 所示。

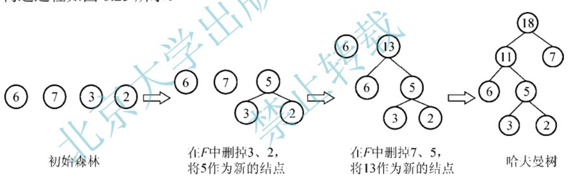


图 6.25 哈夫曼树的构造过程

2. 哈夫曼树的存储结构

由哈夫曼树的构造过程可以得出：哈夫曼树中只有度为 2 的节点，根据二叉树的性质 3 可知，一棵有 n 个叶子节点的哈夫曼树共有 $2n-1$ 个节点。可以采用一个一维数组存储二叉树的所有节点，每个节点需要存储其孩子节点和双亲节点的存储位置信息。这样就可唯一地表示一棵哈夫曼树。节点的结构如图 6.26 所示。

结点权值	双亲节点下标	左孩子节点下标	右孩子节点下标
weight	parent	lchild	rchild

图 6.26 哈夫曼树节点的结构

下面给出哈夫曼树的存储结构。

```
#define n 4
#define m 2*n-1
```

```
    typedef struct                          /*哈夫曼节点存储结构*/
    {
        float weight;                       /*节点权值*/
        int parent,lchild,rchild;           /*双亲及左右孩子的数组下标*/
    }HafuNode,*HafuTree;
  HafuTree tree[m];
```

当节点没有双亲节点或孩子节点时，其相应的链域值为-1，并且将树中的叶子节点集中存储在前 n 个单元中，后 n-1 个单元存储其余非叶子节点。

3. 哈夫曼树的构造算法

哈夫曼给出了一个构造哈夫曼树的方法，即哈夫曼算法。

算法 6.9 哈夫曼树的构造算法。

算法描述如下。

(1) 初始化哈夫曼树中 $2n-1$ 个节点的指针域及权值域分别为-1。

(2) 将 n 个节点的权值依次输入到前 n 个存储单元中。

(3) 再进行 $n-1$ 次合并，共产生 $n-1$ 个新节点，依次放入 $n-1$ 个存储单元中。每次合并的步骤如下。

① 在当前森林的所有节点中，选取双亲为-1 且权值最小的两个根节点 r_1，r_2，将 r_1，r_2 的权值和作为新的节点 r，并将 r 放在后 $n-1$ 个存储单元中。

② 修改 r 的 lchild 和 rchild 域分别为 r_1 和 r_2 的下标值，相应地修改 r1、r2 的 parent 域的值为 r 的下标值。


```
  void HuffMan(HafuTree tree[])
  {  int i,j,p1,p2;
    float small1,small2,f;
    for  (i=0;i<m;i++)                      /*初始化*/
    { tree[i].parent=-1;
     tree[i].lchild=-1;
     tree[i].rchild=-1;
     tree[i].weight=0.0;
    }
  for(i=0;i<n;i++)                          /*读入前 n 个节点的权值*/
  {scanf("%f",&f);
     tree[i].weight=f;
  }
  for  (i=n;i<m;i++)                        /*进行 n-1 次合并,产生 n-1 个新节点*/
  {p1=0;p2=0;
     small1=maxval; small2=maxval;          /*maxval 是 float 类型的最大值*/
     for  (j=0;j<i-1;j++)                   /*选出两个权值最小的根节点*/
    if(tree[j].parent==0)
     if(tree[j].weight<small1)
  {small2=small1;                           /*改变最小权、次小权及对应的位置*/
```

```
small1=tree[j].weight;
p2=p1;
p1=j;
}
else
if(tree[j].weight<small2)
  {small2=tree[j].weight;          /*改变次小权及位置*/
  p2=j;
}
tree[p1].parent=i+1;
tree[p2].parent=i+1;
tree[i].lchild=p1+1;          /*最小权根节点是新节点的左孩子,分量号是下标加 1*/
tree[i].rchild=p2+1;          /*次小权根节点是新节点的右孩子*/
tree[i].weight=tree[p1].weight+tree[p2].weight;
}
}/*Huffman*/
```

例 6.5 已知权值集合为{6，2，3，9，12，24，10，8}，根据哈夫曼树的构造算法，构造一棵哈夫曼树树，并给出结点数组的初始状态和最终状态，如图 6.27 所示。

下标	weight	parent	lchild	rchild
0	6	−1	−1	−1
1	2	−1	−1	−1
2	3	−1	−1	−1
3	9	−1	−1	−1
4	12	−1	−1	−1
5	24	−1	−1	−1
6	10	−1	−1	−1
7	8	−1	−1	−1
8				
9				
10				
11				
12				
13				
14				

(a) 初始状态

下标	weight	parent	lchild	rchild
0	6	−1	−1	−1
1	2	−1	−1	−1
2	3	−1	−1	−1
3	9	−1	−1	−1
4	12	−1	−1	−1
5	24	−1	−1	−1
6	10	−1	−1	−1
7	8	−1	−1	−1
8	5	9	2	1
9	11	11	0	8
10	17	12	3	7
11	21	13	9	6
12	29	14	10	4
13	45	14	5	11
14	74	−1	13	12

(b) 最终状态

图 6.27 哈夫曼树结点数组的初始状态和最终状态

因为 n=8，所以数组的总长度为 $2n$−1=15，根据算法描述可知，数组的初始状态只有叶子结点，图中的-1 表示其双亲或者孩子结点为空。最终状态则为构造的哈夫曼树。若相应的权值集合对应的字符集合为{A、B、C、D、E、F、G、H}，则哈夫曼树如图 6.28 所示。

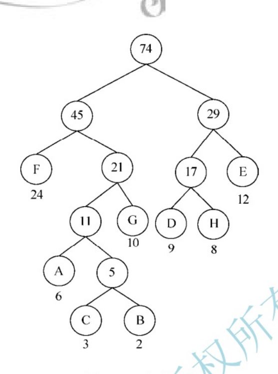


图 6.28　哈夫曼树

6.5.3　哈夫曼树的应用

哈夫曼编码是数据压缩技术中的一种无损压缩方法。在数据通信中，需要对通信信息以二进制的 0、1 进行编码。例如，需要传送的报文为“ABACDDCA”，则传送前先将其进行编码，即将报文中的字符转换成二进制的 0、1 序列。报文中仅含有四种字符，因此只需要长度为 2 的二进制串便可识别。

若 A、B、C、D 这四个字符的编码分别为 00、01、10、11，则上述七个字符的报文可翻译成“0001001011111000”，报文总长度为 14 位，对方接收到报文后，可按照二位一个部分进行译码。但是，在进行报文传送时，总是希望码长越短越好。如果对每个字符设计长度不等的编码，并且让出现次数较多的字符编码尽可能短，则总的码长便可减少。若对于字符 A、B、C、D 采用如下编码：0，00，1，01，则上述报文可翻译成“000011010”，报文总长度为九位。但是，在接收方进行译码的时候就会出现问题，如前 4 位“0000”可以翻译成“AAAA”或“AAB”或“BB”等。因此，在设计长短不等的编码时，必须满足：每个字符的编码都不是另一个字符编码的前缀，称这种编码为前缀编码。

1. 哈夫曼编码

二叉树可以实现字符的前缀编码。若将字符节点表示为叶子节点，节点上的权值表示节点出现的次数，以此就可设计一棵哈夫曼树，且约定二叉树中的左分支为 0、右分支为 1，则每个字符的编码可以表示为从根节点到叶子节点所经过的分支构成的 0、1 序列。通过哈夫曼树得到的二进制前缀编码又称为哈夫曼编码。如果上面报文中 A、B、C、D 的频度分别为 0.3，0.1，0.2，0.2，则其对应的哈夫曼树如图 6.29 所示。

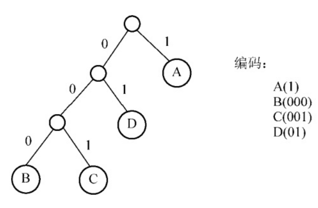


图 6.29　报文“ABACDDCA”对应的哈夫曼树

相应的哈夫曼编码可以表示为1000100101010011。

2. 哈夫曼编码的算法实现

算法如下。

```
    Typedef  char *HuffmanCode;          /*动态分配数组存储哈夫曼编码*/
     void Huffmancode(HafuTree Htree, HuffmanCode HC,int n)
    /*根据哈夫曼树求出哈夫曼编码*/
    {  int c,p,start;char *cd;
       HC=(HuffmanCode)malloc((n+1)*sizeof(char*));   /*求出的哈夫曼编码*/
       cd=(char*)malloc(n*sizeof(char));
       cd[n-1]= '\0';
       for(int i=0;i<n;i++)
       {
           start=n;
           c=i+1;                         /*从叶子节点出发向上回溯*/
           p=Htree[i].parent;             /*Htree[p-1]是Htree[i]的双亲*/
           while(p!=0)
           {
               start--;
               if(Htree[p-1].lchild==c)
                   cd[start]='0';         /*Htree[i]是左子树,生成代码'0'*/
               else
                   cd[start]='1';         /*Htree[i]是右子树,生成代码'1'*/
                   c=p;
                   p=Htree[p-1].parent;
       }
           HC[i]=(char*)malloc((n-start)*sizeof(char)); /*第 i+1 个字符的编
码存入HC[i]*/
           strcpy(HC[i],&cd[start]);
       }
    free(cd);//释放工作空间
    }/*Huffmancode*/
```


若以图6.29所示的哈夫曼树为例，则上述算法求出的哈夫曼编码如图6.30所示。

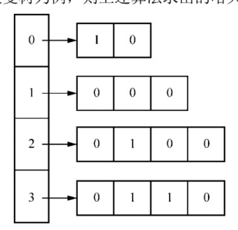


图6.30　哈夫曼编码的动态数组存储

6.5.4　用哈夫曼树解决文件传输编码问题

(1) 定义哈夫曼树中数据元素类型。

```
typedef struct                    /*哈夫曼节点存储结构*/
{
    char ch;                      /*节点字符*/
    int w;                        /*节点权值*/
    int lchild,rchild;            /*左右孩子的数组下标*/
}HafuNode,*HafuTree;

HafuTree ht;                      /*声明一个指向树节点的指针*/

typedef struct
{
    char ch;                      /*叶子节点字符*/
    char codestr[20];             /*字符编码*/

}HafuCode;

HafuCode code[27];                /*用于存放对应字符到哈夫曼编码*/
```

(2) 编写程序实现各项功能

```
#include <stdio.h>
#include <stdlib.h>                       /*其他库函数声明*/

int num;                                  /*记录节点数*/
int codenum=0;                            /*已经获得的编码个数*/
char filename[20]="";                     /*存储文件名*/

void InitHafuArry()
/*导入文件计算权值,生成只含有叶子节点的 HafuNode 数组*/
{
    int j,i,k;
    HafuNode tmpht;
    FILE *fp = NULL;                      /*定义一个指向打开文件的指针*/
    char ch;                              /*用于存储一个字母*/
    char location[30]="D:\\";

    ht = (HafuTree)malloc(53*sizeof(HafuNode)); /*为哈夫曼树分配内存空间*/

    if(ht == NULL)
        return ;

    for(i=0; i < 53; i++)          /*初始化所有的数据单元,每个单元自成一棵树*/
    {
        ht[i].w=0;                 /*权值初始化为 0*/
        ht[i].lchild  = ht[i].rchild=-1;     /*左右子树为空*/
    }

    num=0;

    printf("File name:");
    scanf("%s",filename);
    strcat(location,filename);
    fp=fopen(location,"r");
```

```
        if(!fp)                              /*返回1时即存在文件*/
        {
            printf("Open Error");
            return;
        }

        while(!feof(fp))                     /*没到结尾时返回0*/
        {
            ch=fgetc(fp);
            if(ch == ' ' || ch <= 'z' && ch >= 'a' || ch <= 'Z' && ch >= 'A')
            {
                printf("%c",ch);
                if(ch == ' ')
                    ch='#';

                for(j = 0; j < num; j++)
                {
                    if(ht[j].ch == ch)
                    {
                     break;
                    }
                }

                if(j == num)                 /*找到新字符*/
                {
                    ht[num].ch = ch;         /*将新字符存入并将权值相加1*/
                    ht[num].w++;
                    num++;
                }
                else
                {
                    ht[j].w++;               /*将已有字符权值相加1*/
                }
            }
        }
        fclose(fp);
        printf("\n");
        for(i = 0;i < num; i++)              /*对叶子节点按权值进行升序排序*/
        {
            k=i;

            for(j = i+1; j < num; j++)
            {
                if(ht[j].w<ht[k].w)/*如果后面发现权值比 i 小,则将其下标记录下来,
循环完成之后找到最小的*/
                    k=j;
            }
            if(k!=i)  /*如果权值最小的不是第i个元素,则交换位置,将小的放到前面*/
```

```
            {
                tmpht=ht[i];
                ht[i]=ht[k];
                ht[k]=tmpht;
            }
        }
    }

    int CreateHuffman(HafuTree ht)
     /*在数组中生成哈夫曼数,返回根节点下标*/
    {
        int i,k,j,root;
        HafuNode hfnode;
        codenum=0;
        for(i=0;i<num-1;i++)
        { /*需生成 num-1 个节点*/
            k=2*i+1;                    /*每次取最前面两个节点,其权值必定最小*/
            hfnode.w=ht[k].w+ht[k-1].w;
            hfnode.lchild=k-1;
            hfnode.rchild=k;
            for(j=num+i;j>k;j--)        /*将新节点插入到有序数组中*/
            {
                if(ht[j].w>hfnode.w)
                {
                    ht[j+1]=ht[j];
                }
                else break;
            }
            ht[j]=hfnode;
            root=j;                   /*一直跟随新生成的节点,最后新生成的节点为根节点*/
        }
        return root;
    }

    void GetHafuCode(HafuTree ht,int root,char *codestr)
    /*ht 是哈夫曼树,root 是根节点下标,codestr 是用来暂时存放叶子节点编码的,一开始为空*/
    {
        FILE *out;
        int len,i;
        FILE *fp;                     /*定义一个指向打开文件的指针*/
        char ch;                      /*用于存储一个字母*/
        char location[30]="D:\\";
        if(ht[root].lchild==-1)
        {/*遇到递归终点是叶子节点的,记录叶子节点的哈夫曼编码*/
            code[codenum].ch=ht[root].ch;
            strcpy(code[codenum].codestr,codestr);
            codenum++;
```

```
        }
        else                         /*不是终点则继续递归*/
        {
            len=strlen(codestr);
            codestr[len]='0';     /*左分支编码 0*/
            codestr[len+1]=0;     /*向左孩子递归之前调整编码序列末尾加 0,相当于加了
一个'\0'(NULL),其十进制值是 0,以便下次循环时添加字符,否则会被覆盖*/
            GetHafuCode(ht,ht[root].lchild,codestr);     /*向左递归*/
            len=strlen(codestr);
            codestr[len-1]='1';  /*右分支编码为 1,向右递归之前末尾编码 0 改为 1*/
            GetHafuCode(ht,ht[root].rchild,codestr);     /*向右递归*/
            len=strlen(codestr);
            codestr[len-1]=0;             /*左右孩子递归返回后,删除编码标记末尾*/
        }
        strcat(location,filename);
        fp=fopen(location,"r");
        if(!fp) /*返回 1 时即存在文件*/
        {
            printf("Open Error");
            return;
        }
        out=fopen("D:\\code.txt","w+");
        if(!out)
        {
            /*printf("Write Error");  */
            return;
        }
        while(!feof(fp))              /*没到结尾时返回 0*/
        {
            ch=fgetc(fp);             /*重新打开源文件,对照哈夫曼编码译成编码*/
            if(ch==' '||ch<='z'&&ch>='a'||ch<='Z'&&ch>='A')
            {  if(ch==' ') ch='#';        /*如果是空格则用#号代替*/
            for(i=0;i<codenum;i++)
            {                             /*找到字符所对应的哈夫曼编码*/
                if(ch==code[i].ch)
                {                         /*将所得哈夫曼编码输出到文件中*/
                    fputs(code[i].codestr,out);
                }
            }
            }
        }
        fclose(fp);                   /*关闭打开的两个文件*/
        fclose(out);
    }

    void DecodeHuffmanCode(HafuTree ht,int root)
```

```
/*将哈夫曼编码翻译为明文*/
{
    FILE *fp2;                  /*定义一个指向打开文件的指针*/
    char ch;                    /*用于存储一个字母*/
    int curr=root;              /*当前节点到下标*/
    char filename2[20]="";      /*获得文件名*/
    char location[30]="D:\\";
    printf("File name:");
    scanf("%s",filename);
    strcat(location,filename);
    fp2=fopen(location,"r");
    if(!fp2)                    /*返回 1 时即存在文件*/
    {
        printf("Open Error2");
        return;
    }
    printf("Code:");
    while(!feof(fp2))           /*没到结尾时返回 0*/
    {
        ch=fgetc(fp2);
        if(ch>='0'&&ch<='1') /*将编码过滤出来*/
        {
            printf("%c",ch); /*将密文输出显示*/
        }
    }
    printf("\n");

    rewind(fp2);                /*将文件指针位置定位到开头*/
    while(!feof(fp2))           /*没到结尾时返回 0*/
    {
        ch=fgetc(fp2);
        if(ch>='0'&&ch<='1') /*将编码过滤出来*/
        {
            if(ch=='0')         /*如果为 0 则当前节点向左走*/
            {
                if(ht[curr].lchild!=-1)
                {
                    curr=ht[curr].lchild;   /*若有左子树,则遍历左子树*/
                }
                else
                {
                    curr=root;              /*没有则返回根节点*/
                }
```

```
            }
            if(ch=='1')                        /*如果为1,则当前节点向右遍历*/
            {
                if(ht[curr].rchild!=-1)
                {
                    curr=ht[curr].rchild;   /*若有右子树,则遍历右子树*/
                }
                else
                {
                    curr=root;              /*没有则返回根节点*/
                }
            }
            if(ht[curr].lchild==-1&&ht[curr].rchild==-1)/*若为叶子节点,
则打印输出*/
            {
                printf("%c",ht[curr].ch=='#'?' ':ht[curr].ch);
                curr=root;                  /*回到根节点继续索引*/
            }
        }
    }
    fclose(fp2);
}

void main()
{
    int root;
    char codestr[20]="";
    int control;

    /*显示菜单可选择编码、解码还是退出*/
    printf("===============Menu==============\n");
    printf(" 1:  编        码  \n");
    printf(" 2:  解  码(先编码)\n");
    printf(" 3:  退        出  \n");
    printf(" 请选择一个序号: \n");

    scanf("%d",&control);
    while(control!=3)                       /*只要没有选择退出就一直循环*/
    {
        FILE *output;
        char ch;
        switch (control)
```

```
{
case 1:                                /*选择编码选项*/

        InitHafuArry();                /*初始化节点*/
        root=CreateHuffman(ht);        /*构造一棵哈夫曼树*/
        GetHafuCode(ht,root,codestr);/*根据哈夫曼树将明文译成密码*/
        printf("Code:");

        output=fopen("D:\\CODE.TXT","r");
        if(!output)                        /*返回 1 时即存在文件*/
        {
            printf("Open Error3");
            continue;
        }
        while(!feof(output))               /*没到结尾则返回 0*/
        {
            ch=fgetc(output);
            if(ch>='0'&&ch<='1')           /*将编码过滤出来*/
            {
                printf("%c",ch);           /*将密文输出显示*/
            }
        }
        fclose(output);                    /*将打开的文件关闭*/
        break;

    case 2:                                /*如果选择解码,则调用解码函数*/
        DecodeHuffmanCode(ht,root);
        break;
    case 3:                                /*如果选择了则退出程序*/
        exit(0);

    default:
        printf(" 请选择一个序号:\n");
        break;

}
/*若没有退出则继续输出菜单提示以供选择*/
printf("===============Menu==============\n");
printf(" 1:  编          码  \n");
printf(" 2:  解  码(先编码)\n");
printf(" 3:  退          出  \n");
printf(" 请选择一个序号: \n");
```

```
            getch();
            scanf("%d",&control);
        }
    }
```

在D盘有一个名称为test.txt的文件，运行程序之后生成的密码文件为code.txt，使用时需先编码再解码。



独立实践

现在的程序要使用必须先编码再解码，如何改进能够使程序直接将已编码的文件解码?

小　　结

树和二叉树是一类具有层次或嵌套关系的非线性结构，被广泛地应用于计算机领域。本章着重介绍了二叉树的概念、性质和存储表示；二叉树的三种遍历操作，线索二叉树的有关概念和运算；同时介绍了树、森林与二叉树之间的转换；树的三种存储表示法，树和森林的遍历法；最后讨论了最优二叉树(哈夫曼树)的概念及其应用。

本章是本书的重点之一，建议读者熟练掌握6.2～6.5各节的内容。熟悉树和二叉树的定义和有关术语，理解和记住二叉树的性质，熟练掌握二叉树的顺序存储和链式存储结构。遍历二叉树是二叉树中各种运算的基础，希望读者能灵活运用各种次序的遍历算法，实现二叉树的其他运算。二叉树线索化的目的是加速遍历过程和有效利用存储空间，希望读者熟练掌握在中根线索树中，查找给定节点的中序前趋和后继方法，并能掌握树和二叉树之间的转换方法，存储树的双亲链表法、孩子链表法和孩子兄弟链表法。最后，建议读者理解树和森林的遍历、最优二叉树的特性。

习　　题

一、填空题

1．有一棵树如图6.31所示，回答下面的问题。

(1) 该树的根节点是________。

(2) 该树的叶子节点是________。

(3) 节点K_3的度是________。

(4) 这棵树的度是________。

(5) 这棵树的深度是________。

(6) 节点K_3的孩子节点是________。

(7) 节点K_3的双亲节点是________。

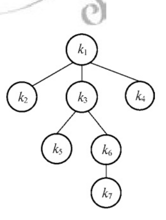


图 6.31　一棵树

2．树和二叉树的三个主要区别是________、________、________。

3．从概念上讲，树与二叉树是两种不同的数据结构，将树转化为二叉树的基本目的是________。

4．一棵二叉树的节点数据采用顺序存储结构，存储于数组 t 中，如图 6.32 所示，则该二叉树的链接表示形式为________。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
e	*a*	*f*		*d*		*g*			*c*	*j*			*l*	*h*						*b*

图 6.32　数组 t

5．深度为 k 的完全二叉树至少有________个节点，至多有________个节点。若按自上而下、从左到右的次序给节点编号(从 1 开始)，则编号最小的叶子节点的编号是________。

6．在一棵二叉树中，度为 0 的节点的个数为 n_0，度为 2 的节点的个数为 n_2，则有 n_0=________。

7．节点最少的树为________，节点最少的二叉树为________。

8．按中根遍历二叉树的结果为 abc，则有________种不同形态的二叉树可以得到这一遍历结果，这些二叉树分别是________。

9．根据如图 6.33 所示的二叉树，回答以下问题。

(1) 其中序遍历序列为________。

(2) 其先根遍历序列为________。

(3) 其后根遍历序列为________。

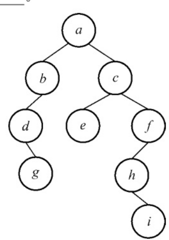


图 6.33　二叉树

二、判断题

1．树最适合用来表示元素之间具有分支层次关系的数据。　　　　　　　　　　　　(　　)

2．二叉树按某种顺序线索化后，任一节点均处在其孩子节点的前面。　　　　　　　(　　)

3．二叉树的先根遍历序列中，任意一个节点均处在其孩子节点的前面。　　　　　　(　　)

4．由于二叉树中每个节点的度最大为2，所以二叉树是一种特殊的树。　　　　　　(　　)

5．树的基本遍历策略可分为先根遍历和后根遍历；二叉树的基本遍历策略可分为先根遍历、中根遍历和后根遍历。人们把由树转化得到的二叉树称为这棵树对应的二叉树，则树的先根遍历序列与其对应的二叉树的先根遍历序列相同。　　　　　　　　　　　　(　　)

三、选择题

1．假设在一棵二叉树中，双分支节点树为15，单分支节点树为30，则叶子节点树为(　　)。

A．16　　B．18　　C．10　　D．12

2．根据二叉树的定义，具有三个节点的不同形状的二叉树有(　　)种。

A．3　　B．5　　C．4　　D．6

3．按照二叉树的定义，具有三个不同数据节点的不同的二叉树有(　　)种.

A．30　　B．5　　C．14　　D．16

4．深度为5的二叉树至多有(　　)个节点。

A．5　　B．35　　C．31　　D．36

5．设高度为 h 的二叉树上只有度为0和度为2的节点，则此类二叉树中所包含的节点树至少为(　　)。

A．$2h$　　B．$2h-1$　　C．$2h+1$　　D．$h+1$

6．任何一棵二叉树的叶子节点在先根、中根和后根遍历序列中的相对次序(　　)。

A．发生改变　　B 不发生改变　　C．不能确定　　D．以上都不对

7．如果二叉树的先根遍历结果为 *stuwv*，中根遍历结果为 *uwtvs*，那么该二叉树的后根遍历序列为(　　)。

A．*uwvts*　　B．*vwuts*　　C．*wuvts*　　D．*wutsv*

8．在一棵非空二叉树的中序遍历序列中，根节点的右边(　　)。

A．只有右子树上的所有节点　　B．只有右子树上的部分节点

C．只有左子树上的部分节点　　D．只有左子树上的所有节点

9．设 a、b 为一棵二叉树上的两个节点，在中根遍历时，a 在 b 前的条件是(　　)。

A．a 在 b 的右方　　B．a 在 b 的左方

C．a 是 b 的祖先　　D．a 是 b 的子孙

10．如果二叉树的后根遍历结果为 *dabec*，中根遍历结果为 *debac*，那么该二叉树的先根遍历序列为(　　)。

A．*acbed*　　B．*decab*　　C．*deabc*　　D．*cedba*

11．根据使用频率为五个字符设计的哈夫曼编码不可能是(　　)。

A．111，110，10，01，00　　B．000，001，010，011，1

C．100，11，10，1，0　　D．001，000，01，11，10



12．设有 13 个值，用它们组成一棵哈夫曼树，则该哈夫曼树共有(　　)个节点。

A．15　　B．25　　C．11　　D．26

四、简答题

1．根据二叉树的定义，具有三个结点的二叉树有 5 种不同的形态，请将它们分别画出。

2．对图 6.35 所示二叉树进行以下操作。

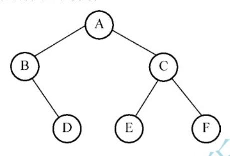


图 6.35　二叉树

(1) 分别画出顺序存储结构和链式存储结构

(2) 写出前、中、后根遍历的序列；

(3) 对该二叉树进行中序线索化；

(4) 并将其转换成树或者森林。

3．假设一棵二叉树的先序序列为 EBADCFHGIKJ，中序序列为 ABCDEFGHIJK，请画出该树。

4．以数据集{4,5,6,7,10,12,18}为结点权值，画出构造 Huffman 树的每一步图示，计算其带权路径长度。

五、编程题

1．编写算法：对于一棵二叉树，统计其叶子节点的个数。

2．编写算法：对于一棵二叉树根节点不变，将其左、右子树进行交换，树中每个节点的左右子树进行交换。

3．假设用于通信的电文仅有八个字母(a、b、c、d、e、f、g、h)组成，字母在电文中出现的频率分别为 0.07、0.19、0.02、0.06、0.32、0.03、0.21、0.10。试为这八个字母设计哈夫曼编码。

4．编写算法，对一棵以孩子兄弟链表表示的树统计叶子的个数。



图



校园电子导航平台

当人们到一个陌生的地方旅游时，可能会找导游为自己在游玩的过程中提供帮助。导游可以提供很多服务，如介绍参观景点的历史背景等相关信息，推荐到下一个景点的最佳路径，解答旅游者所提出的关于旅游景点的相关问询等。对于刚来到校园的新生，对校园环境不熟悉的情况也是如此，一般都是高年级的学生充当“校园导游”的角色。如果能够提供一个校园导航平台，让新生或来访的客人自主地通过与机器的“对话”来获得相关信息，将会节省大量的人力和时间，并且所提供的信息能做到尽可能地准确、详尽。一个成功的校园电子导航平台可以替代现实生活中的“校园导游”，更方便大家查询校园的相关信息。

校园电子导航平台应包含如下的主体功能。

(1) 显示校园平面图，方便用户直观地看到校园的全景示意图，并确定自己的位置。

(2) 为用户提供对平面图中任意场所相关信息的查询。

(3) 为用户提供对平面图中任意场所的问路查询。

7.1 图的定义和术语

图(Graph)是一种较线性表和树更为复杂的数据结构。在线性表中，数据元素之间是被串联起来的，仅有线性关系，每个数据元素只有一个直接前趋和一个直接后继；在树形结构中，数据元素之间有着明显的层次关系，并且每一层上的数据元素可能和下一层中的多个元素相关，但只能和上一层中一个元素相关；而在图形结构中，节点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。

图的定义：图是由顶点的有穷非空集合和顶点之间边的集合组成的，记作 $G=(V(G), E(G))$，简称 $G=(V, E)$。其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合，如图 7.1 所示。

对于图的定义，需要明确以下几个注意的地方。

(1) 线性表中的数据元素称为元素，树中将数据元素称为节点，在图中的数据元素则称为顶点(Vertex)。

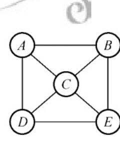


图 7.1　图 G

(2) 线性表中可以没有数据元素，称其为空表；树中可以没有节点，称其为空树；但是，图结构中不允许没有顶点，因为顶点集合 V 是有穷非空集。

(3) 线性表中，相邻的数据元素之间具有线性关系；树结构中，相邻两层的节点间具有层次关系；而图中，任意两个顶点间都可能有关系，顶点之间的逻辑关系用边来表示，边集可以是空的。

7.1.1　各种图定义

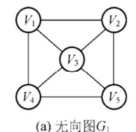
(1) 无向边(Edge)：若顶点 v_i 到 v_j 之间的边没有方向，则称这条边为无向边，用无序偶对(v_i，v_j)表示。

(2) 无向图(Undirected graphs)：任意两个顶点之间的边都是无向边的图。图 7.2(a)就是一个无向图，由于边是无方向的，因此，连接顶点 v_1、v_2 的边可以表示成无序对(v_1，v_2)或(v_2，v_1)，即(v_1，v_2)=(v_2，v_1)。

无向图 G_1 表示为 G_1=(V，E)，其中 V={v_1，v_2，v_3，v_4，v_5}，E={(v_1，v_2)，(v_1，v_4)，(v_2，v_3)，(v_2，v_5)，(v_3，v_4)，(v_3，v_5)}。

(3) 有向边(Arc)：若顶点 v_i 到 v_j 之间的边有方向，则称这条边为有向边，用有序偶对 $<v_i, v_j>$ 表示，v_i 称为弧尾，v_j 称为弧头。

(4) 有向图(Directed graphs)：图中任意两个顶点间的边都是有向边的图。图 7.2(b)就是一个有向图，图中连接顶点 v_1 和 v_2 的有向边可以表示成弧 $<v_1, v_2>$，v_1 是弧尾，v_2 是弧头，且 $<v_1, v_2> \neq <v_2, v_1>$。



(a) 无向图 G_1

(b) 有向图 G_2

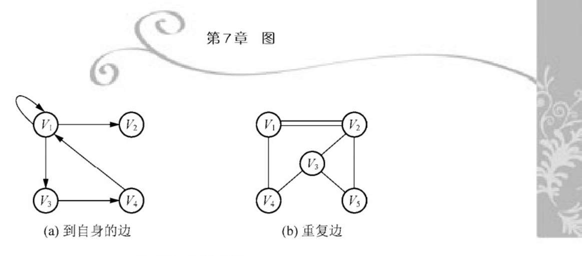
图 7.2　无向图和有向图

有向图 G_2 表示为 G_2=(V，E)，其中 V={v_1，v_2，v_3，v_4}，E={$<v_1, v_2>$，$<v_1, v_3>$，$<v_3, v_4>$，$<v_4, v_1>$}。

注意：无向边用小括号“()”表示，而有向边用尖括号“<>”表示。

(5) 简单图：不存在顶点到其自身的边，且同一条边不重复出现的图。本书中讨论的都是简单图。图 7.3 中的两个图都不是简单图。

(6) 无向完全图：任意两个顶点之间都存在边的无向图。



(a) 到自身的边　　(b) 重复边

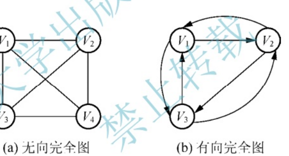
图 7.3　非简单图

性质 1：含有 n 个顶点的无向完全图有 $\dfrac{n\times(n-1)}{2}$ 条边。

证明：任何一个顶点 v_i 都和其他 n−1 个顶点 v_j 有边，则这样的边有 n−1 条；共有 n 个具备该特征的顶点，因此共有 $n\times(n-1)$ 条边；因为有(v_i，v_j)和(v_j，v_i)是重复边，因此总边数除以 2。

图 7.4(a)就是无向完全图，因为每个顶点都要与除它以外的顶点连线，顶点 v_1 和 v_2、v_3、v_4 三个顶点连线，共有四个顶点，边数为 3×4=12，但由于顶点 v_1 与顶点 v_2 连线后，计算 v_2 与 v_1 连线就是重复的，因此整体要除以 2，共有六条边。

(7) 有向完全图：任意两个顶点之间都存在方向相反的两条弧的有向图，如图 7.4(b)所示。



(a) 无向完全图　　(b) 有向完全图

图 7.4　无向完全图和有向完全图

性质 2：含有 n 个顶点的无向完全图有 $n\times(n-1)$ 条边。

证明：任何一个顶点都和其他顶点有边，则这样的边有 n−1 条；共有 n 个具备该特征的顶点，因此有 $n\times(n-1)$ 条边；图中无重复边，因此不用除以 2。

也可以得到结论，对于具有 n 个顶点和 e 条边数的图，无向图有 $0\leqslant e\leqslant n(n-1)/2$，有向图有 $0\leqslant e\leqslant n(n-1)$。

(8) 稀疏图(Spares graph)：有很少条边或弧的图。

稠密图(Dense graph)：相对于稀疏图而言，有很多条边或弧的图。

(9) 权(Weight)：图的弧或边上具有与它相关的数字。

网(Network)：带权的图。图中的权表示从一个顶点到另一个顶点的距离或耗费。图 7.5 就是一张带权的图，即标识中国四大城市的直线距离的网，图中的权就是两地的距离。

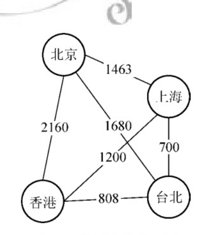


图 7.5 城市距离的网图

(10) 子图(Subgraph)：假设有两个图 $G=(V, E)$和 $G'=(V', E')$，如果 $V'\subseteq V$，且 $E'\subseteq E$，则称 G'为 G 的子图。图 7.6 中的(b)图为左侧无向图和有向图的子图。

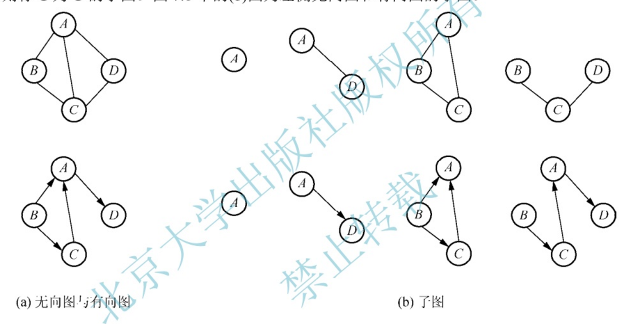


图 7.6 子图

7.1.2 图的顶点与边间关系

1. 邻接与关联

对于无向图 $G=(V, E)$，如果边$(v, v')\in$E，则称顶点 v 和 v'互为邻接点(Adjacent)，即 v 和 v'相邻接。边(v, v')依附于顶点 v 和 v'，或者说边(v, v')与顶点 v 和 v'相关联。

如图 7.6(a)的无向图，顶点 A 与 B 互为邻接点，边(A, B)依附于顶点 A 与 B 上。

对于有向图 $G=(V, E)$，如果弧$<v, v'>\in E$，则称顶点 v 邻接到顶点 v'，顶点 v'邻接自顶点 v。弧$<v, v'>$和顶点 v，v'相关联。

如图 7.6(a)的有向图，顶点 A 邻接到顶点 D，顶点 D 邻接自顶点 A，弧$<A, D>$和顶点 A、D 相关联。

2. 顶点的度

对于无向图，顶点 v 的度是和 v 相关联的边的数目，记做 TD(v)。在图 7.2(a)的无向图

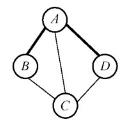
中，顶点 v_1 的度为 3，顶点 v_3 的度为 4，该图的边数是 8，各个顶点度的和=3+3+4+3+3=16，可以得知边数其实就是各顶点度数和的一半，而多出的一半是因为每条边都重复两次计数。因此，$e=\frac{1}{2}\sum_{i=1}^{n}\mathrm{TD}(v_i)$。

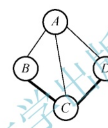
对于有向图，顶点 v 的度 $\mathrm{TD}(v)$分为出度和入度两部分。以顶点 v 为头的弧的数目称为 v 的入度，记为 $\mathrm{ID}(v)$；以顶点 v 为尾的弧的数目称为 v 的出度，记为 $\mathrm{OD}(v)$；顶点 v 的度为 $\mathrm{TD}(v)=\mathrm{ID}(v)+\mathrm{OD}(v)$。

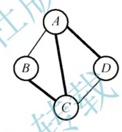
如图 7.2(b)所示，顶点 v_1 的出度为 2(从 v_1 到 v_2 的弧，从 v_1 到 v_3 的弧)，入度为 1(从 v_4 到 v_1 的弧)，所以顶点 v_1 的度为 2+1=3。图中有向图的弧有四条，而各顶点的出度和=2+0+1+1=4，各顶点的入度和=1+1+1+1=4。因此可得，$e=\sum_{i=1}^{n}\mathrm{ID}(v_i)=\sum_{i=1}^{n}\mathrm{OD}(v_i)$。

3. 路径

无向图 $G=(V, E)$中从顶点 v 到 v'的路径是一个顶点序列(v，$v_{i,0}$，$v_{i,1}$，…，$v_{i,m}$，v')，其中($v_{i,j-1}$，$v_{i,j}$)$\in E$，$j\in[1, m]$，即(v，$v_{i,0}$)，($v_{i,0}$，$v_{i,1}$)，…，($v_{i,m}$，v')分别是图中的边。如图 7.7 中，顶点 B 到顶点 D 有四种不同的路径。







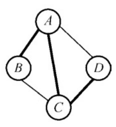
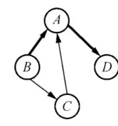


图 7.7　无向图的顶点 B 到顶点 D 的四种路径

有向图 G 的路径也是有向的，顶点序列满足$<v_{i,j-1}, v_{i,j}>\in E$，$j\in[1, m]$。在图 7.8 中，顶点 B 到顶点 D 有两种路径；而顶点 A 到顶点 B 则不存在路径。



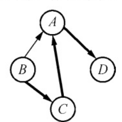


图 7.8　有向图的顶点 B 到顶点 D 的两种路径

树的根节点到任意节点的路径是唯一的，而图中两个顶点之间的路径不是唯一的。

4. 路径长度

路径上的边或弧的数目。图 7.7 的无向图中顶点 B 到顶点 D 的路径长度各不相同，左边两条路径长度为 2，右边两条路径长度为 3。

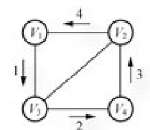
5. 回路、简单路径、简单回路

回路：也称环(Cycle)，指第一个顶点和最后一个顶点相同的路径。

简单路径：序列中顶点不重复出现的路径。

简单回路：也称简单环，指除了第一个顶点和最后一个顶点外，其他顶点不重复出现的回路。

在图 7.9 中，左边的回路(v_1，v_3，v_4，v_2，v_1)由于第一个顶点和最后一个顶点都是 v_1，且 v_3，v_4，v_2 没有重复出现，因此是一个简单回路。而右边的回路(v_1，v_3，v_4，v_2，v_3，v_1)，由于有顶点 v_3 的重复，就不再是简单回路。



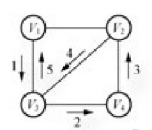


图 7.9　简单回路与非简单回路

7.1.3　连通图的相关术语

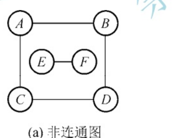
1. 连通

在无向图 G 中，如果从顶点 v 到顶点 v' 有路径，则称 v 和 v' 是连通的。

2. 连通图

如果对于图中任意两个顶点 v_i、$v_j \in V$，若 v_i 和 v_j 都是连通的，则称 G 是连通图(Connected Graph)。

如图 7.10 所示，图(a)的顶点 A 到顶点 B、C、D 都是连通的，但是与顶点 F 或 E 无路径连通，因此该图不是连通图；图(b)中，顶点 v_1、v_2、v_3、v_4 间互相都是连通的，因此是连通图。



(a) 非连通图　　(b) 连通图

图 7.10　非连通图和连通图

3. 连通分量

连通分量指无向图中的极大连通子图。连通分量要求满足以下条件。

(1) 要求是子图。

(2) 子图是连通的。

(3) 连通子图含有极大顶点数，即若增加一个顶点，则其不是连通子图。

(4) 具有极大顶点数的连通子图包含依附于这些顶点的所有边。

如图 7.11 所示，图(a)是一个无向非连通图，但是它有两个连通分量，即图(b)和(c)。而

图(d)尽管是图(a)的子图，但是不满足连通子图的极大顶点数要求，因此不是图(a)的连通分量。

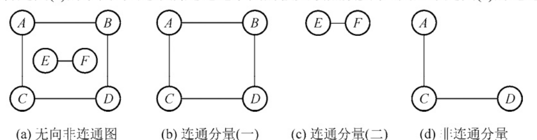


图 7.11 连通分量

4. 强连通图

在有向图 G 中，如果对于每一对 v_i、$v_j \in V$，$v_i \neq v_j$，从 v_i 到 v_j 和从 v_j 到 v_i 都存在路径，则称 G 是强连通图。

5. 强连通分量

强连通分量指有向图中的极大强连通子图。

在图 7.12 中，图(a)不是强连通图，因为顶点 A 到顶点 D 存在路径，而 D 到 A 不存在路径。图(b)是强连通图，而且是图(a)的极大强连通子图，即图(a)的强连通分量。

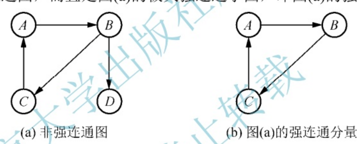


图 7.12 强连通分量

6. 连通图的生成树

连通图 G 的一个极小连通子图，包含有 G 中所有 n 个顶点，但是只有足以构成一棵树的 n-1 条边。

生成树有以下三个要素。

(1) G 中的全部 n 个顶点。

(2) 图为连通图。

(3) 只有 n-1 条边。

在图 7.13 中，(a)是一个普通图，显然不是生成树，因为其顶点数为 8，而边数为 9，当去掉两条构成环的边后，如图(b)，图(c)所示，就满足了 n 个顶点 n-1 条边且连通的定义，即(b)和(c)都是生成树。由此可知，如果一个图中有 n 个顶点和小于 n-1 条边，则其是非连通图，如果它多于 n-1 条边，则构成一个环，因为这条边使得它依附的两个顶点间有了第二条路径。如图(b)和图(c)中随便在哪两个顶点间加一条边都将构成环。但是有 n-1 条边并不一定是生成树，如图(d)所示，因为图(d)不是连通图。

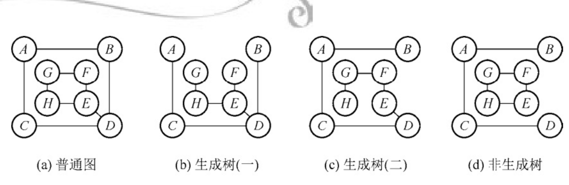


图 7.13　生成树

7. 有向树

若有向图只有一个顶点的入度为 0，其余顶点的入度均为 1，则其称为有向树。所谓的入度为 0 的顶点相当于树中的根节点，其他入度为 1 的顶点是指树的非根节点的双亲只有 1 个。

8. 有向图的生成森林

有向图的生成森林是指由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。

在图 7.14 中，图(a)是一个有向图，去掉一些弧后，可以分解成两棵有向树，如图(b)和(c)，这两棵就是图(a)有向图的生成森林。

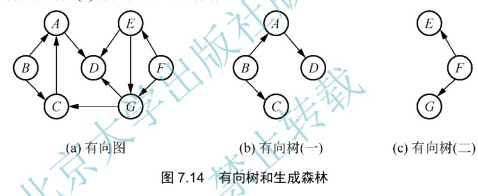


图 7.14　有向树和生成森林

7.2　图的存储结构

图的信息主要有顶点信息和边的信息两部分，因此研究图的存储结构主要研究这两部分信息如何在计算机内表示。

图的存储结构有多种，与线性表和树相比更加复杂，其中最常用的是邻接矩阵和邻接表。

7.2.1　邻接矩阵存储

图的邻接矩阵(Adjacency Matrix)存储方式就是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组 $n \times n$ 阶矩阵存储图中的边或弧的信息，即矩阵元素 a_{ij} 的值表示为顶点 v_i(行)与顶点 v_j(列)间的关系。

设图 $G=(V, E)$具有 $n(n\geq 1)$个顶点 v_1，v_2，…，v_n，和 m 条边或弧 e_1，e_2，…，e_m，则 G 的邻接矩阵是 $n \times n$ 阶矩阵，记为 $A(G)$。其每一个元素 a_{ij} 定义如下。

对于有向图的邻接矩阵来说，当$<v_i, v_j>$是该有向图中的一条弧时，$a_{ij}=1$，否则 $a_{ij}=0$。第 i 个顶点的出度为矩阵中第 i 行中“1”的个数，入度为第 i 列中“1”的个数，并且有向弧的条数等于矩阵中“1”的个数。

对于无向图的邻接矩阵来说，当(v_i, v_j)是该无向图中的一条边时，$a_{ij}=a_{ji}=1$，否则$a_{ij}=a_{ji}=0$。第 i 个顶点的度为矩阵中第 i 行中“1”的个数或第 i 列中“1”的个数。图中边的数目等于矩阵中“1”的个数的一半，因为每条边在矩阵中描述了两次，即

$$a_{ij}=\begin{cases}1 & \text{顶点}v_i\text{与}v_j\text{相邻接}\\ 0 & \text{其他}\end{cases}$$

对于有权值的网图来说，有

$$a_{ij}=\begin{cases}\omega_{ij} & \text{顶点}v_i\text{与}v_j\text{相邻接}\\ \infty & \text{其他}\end{cases}$$

图 7.2(a)无向图 G_1 的邻接矩阵如图 7.15(a)所示，图 7.2(b)有向图 G_2 的邻接矩阵如图 7.15(b)所示。图 7.15(c)所示的权网 G_3 的邻接矩阵如图 7.15(d)所示。

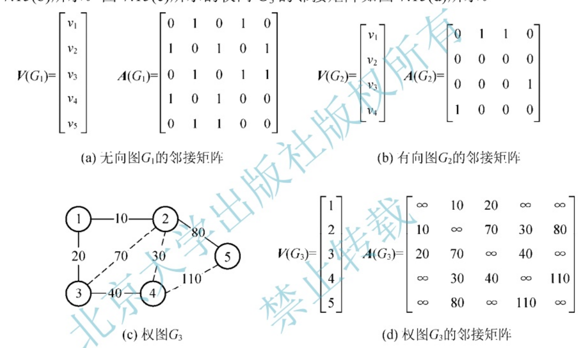


图 7.15　邻接矩阵

邻接矩阵具有如下特征。

(1) 图中各顶点的序号确定后，图的邻接矩阵是唯一确定的。

(2) 无向图和无向网的邻接矩阵是一个对称矩阵，可以压缩存储其下三角矩阵，故 n 个顶点的图只需使用 $n(n+1)/2$ 个存储单元；有向图的邻接矩阵不一定对称，故 n 个顶点需 n^2 个存储单位。

(3) 无向图中顶点 v_i 的度是邻接矩阵中第 i 行(或第 i 列)的非 0 元素的个数。

(4) 有向图中顶点 v_i 的度是邻接矩阵中第 i 行与第 i 列非 0 元素个数之和。

(5) 无向图的边数等于邻接矩阵中非 0 元素个数之和的一半，有向图的弧数等于邻接矩阵中非 0 元素之和。

邻接矩阵表示法对于以图的顶点为主的运算比较适用。此外，除完全图外，其他图的邻接矩阵如果有许多零元素，特别是稀疏图，当 n 值较大，而边数相对很少时，采用邻接矩阵存储图信息就会浪费存储空间。

建立带权无向图邻接矩阵的算法如下。

假设权值为 int 型，每个顶点存放 int 型的顶点编号，首先输入图的顶点数和边数；然后输入顶点编号来建立顶点信息表，并将邻接矩阵中的各元素初始化为 0 或无穷大数；最后按顶点顺序输入每条边的顶点编号和权值，从而建立图的邻接矩阵。

算法 7.1 无向网图的邻接矩阵的创建。

```
#define  MAXSIZE  100                    /*图的顶点个数,由用户确定*/
typedef Char DataType;
typedef struct
 {
   DataType vexs[MAXSIZE];               /*顶点信息表*/
   int edges[MAXSIZE][ MAXSIZE];         /*邻接矩阵*/
   int  numVexs,numEdges ;               /*顶点数和边数*/
 }Graph;

void  Create_Graph(Graph  *G)
 {
   int  i,j ,k,w;
   scanf (“%d”,&(G-> numVexs),& (G -> numEdges));  /*输入顶点数及边数*/
   printf ("请输入顶点信息(顶点编号),建立顶点信息表:\n") ;
   for(i = 0;i< G -> numVexs;i++)
      scanf("%c",&( G ->vexs[i]));      /*输入顶点信息*/
   for (i = 0;i< G -> numVexs;i++)      /*邻接矩阵初始化*/
      for (j = 0;j< G -> numVexs;j++)
          G ->edges[i][j] = 0;
   for (k = 0;k< G -> numEdges;k++)   /*读入边的顶点编号和权值,建立邻接矩阵*/
   {  printf ("请输入第%d条边的顶点序号 i,j 和权值 w:",k+1);
      scanf ("%d,%d,%d",&i,&j,&w);
      G ->edges[i][j] = w;
      G ->edges[j][i] = w;
    }
 }
```

该算法的执行时间是 $O(n+n^2+e)$ ，由于 $e<n^2$ ，所以算法的时间复杂度为 $O(n^2)$ 。

7.2.2 邻接表存储

邻接表(Adjacency List)是图的一种链式分配的存储结构。在邻接表表示法中，用一个顺序存储区来存储图中各顶点的数据，并对图中每个顶点 v_i 建立一个单链表(称为 v_i 的邻接表)，把顶点 v_i 的所有相邻顶点，即其后继顶点的序号连接起来。

邻接表的处理方法如下。

(1) 图中顶点用一个一维数组存储，另外，顶点数组中的每个数据元素还要存储指向第一个邻接点的指针，以便于查找该顶点的边信息。如图 7.16(a)所示，顶点数组(或顶点表)中的每个节点由 data 和 first 两个域表示，其中，数据域 data 用于存储顶点的信息，指针域

first 用于指向边表的第一个节点，即该顶点的第一个邻接点。

(2) 图中每个顶点 v_i 的所有邻接点构成一个线性表，称为边表，由于邻接点个数不一定，因此用单链表存储。如图 7.16(b)中，每个边表节点由 vertex 和 next 两个域组成，其中，邻接点域 vertex 用于存储顶点 v_i 的某个邻接点在顶点表中的序号(或数组下标)，指针域 next 用于指向下一个边表节点。



(a) 顶点表结点　　(b) 边表结点

图 7.16　邻接矩阵表示的节点结构

因此，在无向图的邻接表中，顶点 v_i 的每个边表节点都对应于与 v_i 相关联的一条边，将该邻接表称为边表；而在有向图的邻接表中，顶点 v_i 的每一个边表节点对应于以 v_i 为弧尾的一条弧，因此将该邻接表称为出边表。

如图 7.17 所示的无向图 G_4，其顶点 v_1 的邻接表中两个边表节点 v_2 和 v_3 在顶点表中的序号(或下标)分别为 1 和 2，表示关联与 v_1 的边有两条：(v_1，v_2)和(v_1，v_3)。

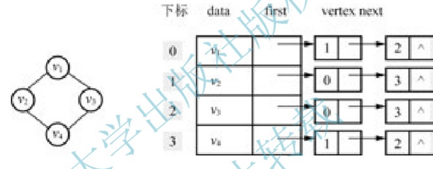
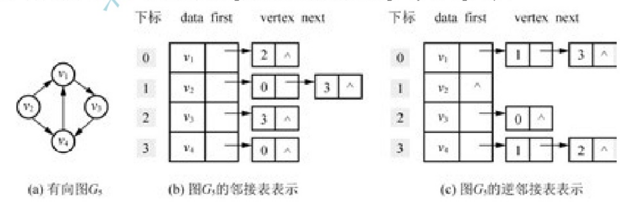


图 7.17　无向图 G_4 的邻接表表示

如图 7.18 所示，有向图 G_5 的邻接表中，顶点 v_2 的邻接表中有两个边表节点，其顶点序号分别为 0 和 3，表示以顶点 v_2 为弧尾的有两条弧<v_2，v_1>和<v_2，v_4>。



(a) 有向图 G_5　　(b) 图 G_5 的邻接表表示　　(c) 图 G_5 的逆邻接表表示

图 7.18　有向图 G_5 的邻接表和逆邻接表表示

若无向图 G 有 n 个顶点、e 条边，则邻接表需 n 个顶点表节点和 $2e$ 个边表节点，每个节点有两个域。显然，对于边很少的图，用邻接表比用邻接矩阵更节省存储单元。

在无向图的邻接表中，第 i 个边表中的节点数就是顶点 v_i 的度数。在有向图的邻接表中，第 i 个边表中的节点数是顶点 v_i 的出度。若要求 v_i 的入度，则必须对邻接表进行遍历，

以统计顶点的值为 i 的边表节点的数目，这样做很费时。为了便于确定有向图中顶点的入度，可另外建立一个逆邻接表，将顶点 v_i 的每个边表节点对应于以 v_i 为弧头的一条弧，即用边表节点的邻接点域 next 存储邻接到 v_i 的顶点的序号。逆邻接表的边表称为入边表。如图 7.18(c)所示即为有向图 G_5 的逆邻接表表示，顶点 v_3 的入边表中有一个边表节点 v_1，表示以 v_3 为弧头的弧有一条$<v_1，v_3>$。

对于权网，则只需在边表节点的结构中增设一个权值域 weight 用于存放权值信息。

邻接表与邻接矩阵之间有如下的关系。

(1) 邻接表(或逆邻接表)表示中，每个边表对应邻接矩阵的一行(或一列)。

(2) 边表中顶点的个数等于该行(或列)中非零元素的个数。

(3) 邻接表中的每个节点对应邻接矩阵中该行的一个非零元素。

(4) 邻接表的顶点节点对应邻接矩阵该行的顶点。

邻接表存储结构类型定义的 C 语言表示如下。

```
#define  NMAX  100                    /*假设顶点的最大数为 100*/
typedef struct edgenode {             /*边表节点类型*/
    int   vertex ;
    struct  edgenode  *next ;
    } EdgeNode, *pointer;

typedef  struct {                     /*顶点表节点类型*/
    DataType  data;
    EdgeNode *first;                  /*边表头指针*/
}HeadType;

typedef  struct {                     /*表头节点向量,即顶点表*/
    HeadType   adlist[NMAX];
    int  numVexs,numEdges ;           /*图中当前顶点数和边数*/
    }LKGraph;
```


建立无向图邻接表的算法如下。

假设每个顶点存放的是一个字符，先输入表头数组的顶点信息 data，并将每个表头的 first 置为 NULL；然后读入顶点对(i，j)，生成两个边表节点，其 vertex 域分别置为 i 和 j，再将它们用头插入法分别插入到第 i 个和第 j 个边表中。在顶点对输入过程中，自动累计边数，若输入的顶点号 $i<0$，则结束。

算法 7.2 无向图的邻接表的创建。

```
void CreatGraph(LKGraph *G)       /*建立无向图的邻接表*/
 {  int  i,j,e,k;pointer  p;
printf(“请输入顶点数：\n”);
scanf (“%d”, &( G -> numVexs;));
for (i =1; i<= G -> numVexs; i++)
{    /*读入顶点信息,建立顶点表*/
    scanf (“ \n %c”, &( G ->adlist [i] . data) );
```

```
        G ->adlist [i] . first = NULL;    }
    e = 0;
    scanf ( "\n%d,%d\n" , &i,&j );        /*读入一个顶点对号 i 和 j*/
    while (i>0)
    {/*读入顶点对号,建立边表*/
        e++; /*累计边数 */
        p = (EdgeNode*)malloc(size(struct edgenode)); /*生成新的邻接点序号为 j
的表节点*/
        p-> vertex = j;
        p->next = G ->adlist[i]. first;
        ga->adlist[i].first = p;     /*将新表节点插入到顶点 vi 的边表的头部*/

        p = (pointer)malloc(size(struct node)); /*生成邻接点序号为 i 的表节点*/
        p-> vertex = i;
        p->next = G ->adlist[j].first;
        G ->adlist[j].first=p;          /*将新表节点插入到顶点 vj 的边表头部*/
        scanf ( "\n%d,%d\n" , &i,&j );  /*读入一个顶点对号 i 和 j*/
        }
        G -> numEdges = e ;
    }
```

对上述建立无向图的邻接表算法略加修改，即将虚线部分中的两段代码选择一段使用，因为对于无向图来说，一条边都对应两个顶点，因此在循环中，一次要对 i 和 j 分别进行插入操作。

建立无向图的邻接表算法的时间复杂度是 $O(n+e)$ 。在邻接表的边链表中，各个表节点的链入顺序任意，视表节点输入次序而定。

7.3 图的遍历

与树的遍历类似，图的遍历是图的运算中最重要的运算，图的许多运算均以遍历为基础，如求连通分量、求最小生成树和拓扑排序等。

从图中某个顶点出发访问图中所有顶点，且使得每个顶点仅被访问一次，这一过程称为图的遍历。根据访问路径方向的不同，主要有两种遍历图的方法：深度优先搜索法(Depth First Search，DFS)和广度优先搜索法(Breadth First Search，BFS)。

因为图的任一个顶点都可能和其他的顶点相邻接，采用不同的搜索顺序，可能出现同一个顶点被访问多次的情况，因此，必须在图遍历的过程中记住每个被访问过的顶点，一般可以设一个数组 visited[i]，设初值为 0 或“假”表示该顶点未被访问。如果该顶点已访问过，则修改 visited[i]的值为 1 或“真”。

书中算法介绍以无向图为例，但是算法思想和实现也适用于有向图。

7.3.1 深度优先搜索遍历

深度优先搜索遍历类似于树的前序遍历，是树的前序遍历的推广。

假设给定图 G 的初态是所有顶点均未曾被访问过，在 G 中任选一个顶点 v 作为初始出发点，则深度优先搜索遍历的基本思想是从顶点 v 出发，访问此顶点；然后依次从 v 的未被访问的邻接点出发进行深度优先遍历，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点做起始点，重复上述过程，直至图中所有顶点都被访问到。

对图 7.19 中的无向图 G_6 进行深度优先搜索，具体遍历过程(以下标小的顶点优先访问为原则)如下。

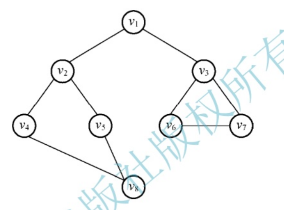


图 7.19 无向图 G_6

(1) 从起始顶点 v_1 出发，将 v_1 标记为已访问。

(2) v_1 有两个邻接点 v_2、v_3，按照小下标先访问的原则，访问邻接顶点 v_2，将 v_2 标记为已访问。

(3) v_2 有三个邻接点 v_1、v_4、v_5，其中 v_1 为已访问顶点，因此访问 v_4，将 v_4 标记为已访问。

(4) v_4 有两个邻接点 v_2、v_8，其中 v_2 为已访问顶点，因此访问 v_8，将 v_8 标记为已访问。

(5) v_8 有两个邻接点 v_4、v_5，其中 v_4 为已访问顶点，因此访问 v_5，将 v_5 标记为已访问。

(6) v_5 有两个邻接点 v_2、v_8，均为已访问顶点，因此退回到 v_8。

(7) v_8 的两个邻接点均为已访问顶点，因此退回到 v_4，以此类推，一直退回到 v_1。

(8) v_1 有两个邻接点 v_2、v_3，其中 v_2 已经访问过，因此访问 v_3，将 v_3 标记为已访问。

(9) v_3 有两个邻接点 v_6、v_7，遵从小下标开始访问原则，访问 v_6，将 v_6 标记为已访问。

(10) v_6 有两个邻接点 v_3、v_7，其中 v_3 为已访问顶点，因此访问 v_7，将 v_7 标记为已访问。

(11) v_7 有两个邻接点 v_6、v_3，均为已访问顶点，因此退回到 v_6。

(12) v_6 的两个邻接点均为已访问顶点，因此退回到 v_3，以此类推，一直退回到 v_1。
至此，所有顶点都是已访问顶点，搜索结束。由此得到的顶点搜索序列为：

$$v_1 \to v_2 \to v_4 \to v_8 \to v_5 \to v_3 \to v_6 \to v_7$$

该序列称为 DFS 序列，一个图的 DFS 序列不一定唯一，这与算法、图的存储结构以及初始出发点有关。而且，由具体遍历过程可看出，深度优先搜索遍历是一个递归过程，其特点是尽可能地先对纵深方向的顶点进行访问。

DFS 算法的步骤如下。

(1) 在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一邻接顶点 w_1。

(2) 从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2。

(3) 从 w_2 出发，进行类似的访问……如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 u 为止。

(4) 退回一步，退到前一次刚访问过的顶点，看是否还有其他没有被访问的邻接顶点。

① 如果有，则访问此顶点，之后从此顶点出发，进行与前述类似的访问，即执行步骤(2)。

② 如果没有，则再退回一步进行搜索，即执行步骤(4)。

重复上述过程，直到图中所有顶点都被访问过为止。

如果采用邻接矩阵来存储图，则深度优先搜索算法的代码实现如下。

算法 7.3　邻接矩阵的深度优先遍历递归。

```
void  DFSM(Graph  *g,int  i )/*邻接矩阵的深度优先递归算法*/
 { int  j;
   printf("深度优先遍历节点:%c\n" ,g->vexs[i]);/*输出顶点*/
   visited[i]=1;                /*将被访问顶点的访问标志设置为 1,表示已访问*/
   for(j =0;j<g->n;j++)
      if((g->edges[i][j] = =1)&& ! visited[j])
         DFSM(g,j);             /*对未访问的邻接顶点递归调用*/
 }

void  DFS(Graph  *g)            /*邻接矩阵的深度遍历操作*/
 {int i;
  for(i=0;i<g->n;i++ )
    visited[i] = 0;             /*初始所有顶点状态是未访问状态*/
  for(i= 0;i<g->n;i++)
    if(!visited[i])
      DFSM(g,i);       /*对未访问过的顶点调用 DFSM,若是连通图,则只执行一次*/
  }
```

如果采用邻接表来存储图，则深度优先搜索算法的代码实现如下：

算法 7.4　邻接表的深度优先遍历递归。

```
void  DFSL(LKGraph  *g,int  n )            /*邻接表的深度优先递归算法*/
 { pointer p;
   printf(" %d\n" ,g-> adlist[n].data);  /*输出顶点*/
   visid[n]=1;                           /*设置访问标志为 1*/
   for(p = g->adlist[n].first;p!=NULL; p = p->next)
      if(! visited[p-> vertex ])
        DFSL(g,p-> vertex );              /*对未访问的邻接顶点递归调用*/
  }

void  DFS(LKGraph  *g)                   /*邻接表的深度遍历操作*/
```

```
    {int i;
     for(i=0;i<g->n;i++ )
        visited[i] = 0;                /*初始所有顶点状态是未访问状态*/
     for(i= 0;i<g->n;i++)
        if(!visited[i])
           DFSL(g,i);  /*对未访问过的顶点调用 DFSL,若是连通图,则只执行一次*/
    }
```

对比两个不同存储结构的深度优先遍历算法，对于 n 个顶点 e 条边的图来说，邻接矩阵由于是二维数组，要查找每个顶点的邻接点需要访问矩阵中的所有元素，因此需要 $O(n^2)$ 的时间。而邻接表做存储结构时，找邻接点所需的时间取决于顶点和边的数量，所以时间复杂度是 $O(n+e)$ 。可见对于点多边少的稀疏图来说，邻接表结构使得算法在时间效率上大大提高了。

7.3.2 广度优先搜索遍历

广度优先搜索遍历类似于树的按层次遍历。设图 G 的初态是所有顶点均未访问过，在 G 中任选一顶点 v_i 为初始出发点，则广度优先搜索的基本思想是：首先访问出发点 v，其次访问 v 的所有邻接点 w_1，w_2，…，w_t，再依次访问与 w_1，w_2，…，w_t 邻接的所有未曾访问过的顶点，以此类推，直至图中所有和初始出发点 v 有路径相通的顶点都已访问过为止。此时，从 v 开始的搜索过程结束，若 G 是连通图则完成遍历。

对图 7.19 中的无向图 G_6 进行广度优先搜索，以顶点 v_1 为起始点，则具体遍历过程如下：访问 v_1 的两个邻接点 v_2、v_3，接着访问 v_2 的两个邻接点 v_4、v_5，再访问 v_3 的两个邻接点 v_6、v_7，继续访问 v_4 的邻接点 v_8，至此，所有顶点都被访问到。因此，得到的顶点搜索序列为 $v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7 \to v_8$。

分析可知，如果顶点 w_1 在顶点 w_2 之前被访问，则访问 w_1 的所有未曾访问过的邻接点之后，再访问 w_2 的尚未被访问的相邻顶点。也就是说，先访问的顶点其邻接点亦先被访问，具有先进先出的特点。因此，可以引进队列保存已访问过的顶点。

BFS 算法的步骤如下。

(1) 从图中某个顶点 v 出发，访问此顶点。

(2) 依次访问 v 的各个未曾访问的邻接点。

(3) 依次从这些邻接点出发再依次访问它们的邻接点。

(4) 直至图中所有和 v 有路径相通的顶点都被访问为止。

(5) 若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点做起始点，重复上述过程，直至图中所有顶点都被访问为止。

如果采用邻接矩阵来存储图，则广度优先搜索算法的代码实现如下。

算法 7.5 邻接矩阵的广度优先搜索。

```
   void BFS(Graph *g,int v)           /*邻接矩阵的广度遍历算法*/
    {int j;
     seqqueue  q;                     /*假设采用顺序队列,定义顺序队列类型变量 q*/
   InitQueue(&q);          /*队列 q 初始化*/
```

```
printf("访问出发点 %d",v);         /*访问出发点,假设为输出顶点序号*/
visited[v] = 1;                    /*置访问标志为1,表示此顶点已访问过*/
EnQueue(&q, v);                    /*顶点 v 入 q 队列*/
while (!QueueEmpty (&q))           /*队列 q 空否?*/
{ DeQueue(&q, &v);                 /*队列 q 非空时,顶点 v 出队*/
  for(j= 1; j<=g-> n; j++)
     if(g->edges [v][j] = = 1 && !visited[j])
         { printf("访问顶点%d",j); visited[j] = 1; /*置顶点 j 被访问标志*/
          EnQueue (&q, j);         /*顶点 j 入队*/
         }
 }
}
```

如果采用邻接表来存储图，则广度优先搜索算法的代码实现如下。

算法 7.6　邻接表的广度优先搜索算法

```
void BFSL(Graph *g,int v)          /*采用邻接表存储结构,BFS 遍历*/
{  SqQueue  q;                     /*假设采用顺序队列,定义顺序队列类型变量 q*/
   pointer  p;
   InitQueue(&q);                  /*队列 q 初始化*/
   printf("访问出发点 %d",v);       /*访问出发点,假设为输出顶点序号*/
   visited[v] = 1;                 /*置访问标志为1,表示此点已访问过*/
   EnQueue(&q, v);                 /*顶点 v（刚访问过的）入队*/
   while (!QueueEmpty (&q))        /*判队列空*/
   { DeQueue(&q, &v);              /*队列 q 不空时,出队*/
     p = g->adlist[v].first;       /*将节点 v 表头指针域存入 p 中*/
     while(p != NULL)
     {          /*访问与顶点 v 相邻接的所有顶点,即以 v 顶点为表头的单链表*/
        If(! visited[p->vertex])
         { printf("%d",p-> vertex);  visid[p-> vertex ]= 1;
           EnQueue(&q, p-> vertex);
         }
        p = p->next;
     }
    }
}
```

对于具有 n 个节点 e 条边的连通图，因为每个顶点均入队一次，所以算法 BFS 和 BFSL 的外循环次数为 n。算法 BFS 的内循环是 n 次，故算法 BFS 的时间复杂度 $O(n^2)$。算法 BFSL 的内循环次数取决于各顶点的边表节点个数，内循环执行的总次数是边表节点的总个数 $2e$，故算法 BFSL 的时间复杂度是 $O(n+e)$。算法 BFS 和 BFSL 所用的辅助空间是队列和标志数组，故它们的空间复杂度为 $O(n)$。

对比图的深度优先遍历和广度优先遍历算法，它们在时间复杂度上相同，不同之处仅在于其对顶点访问的顺序不同，可见两者在全图遍历上没有优劣之分，根据不同情况选择不同算法即可。

7.4　图的生成树

在图论中，常常将树定义为一个无回路连通图。如图 7.20 所示的两个图就是无回路的连通图。尽管看上去它们不是树，但只要选定某个顶点作为根，以根节点为起点对每条边定向，就能将它们变为通常的树。

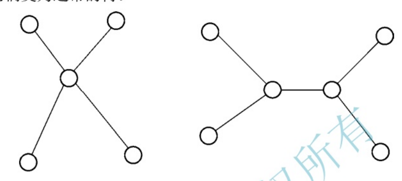


图 7.20　两个无回路的连通图

根据连通图生成树的定义(参考 7.1.3 节)，连通图 G 的生成树是 G 的极小连通子图。所谓极小是指边数最少，若在生成树上任意去掉一条边，就会使之变为非连通图；若在生成树上任意添加一条边，就必定出现回路。

因此，生成树具有如下性质：一个有 n 个顶点的连通图的生成树有且仅有 $n-1$ 条边，一个连通图的生成树并不唯一。

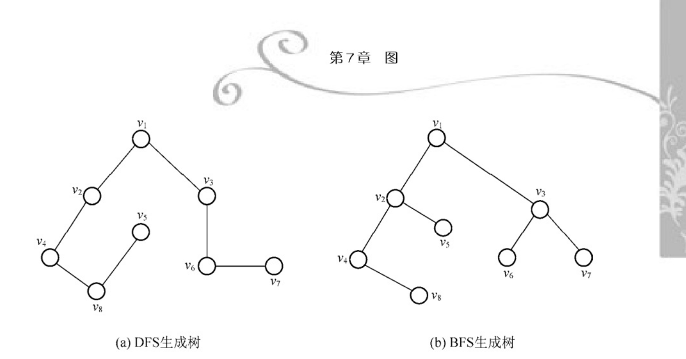
7.4.1　生成树的基本概念

对给定的连通图，如何求得其生成树呢？

设图 $G=(V, E)$是一个具有 n 个顶点的连通图，则从 G 的任一顶点出发，对 G 进行深度优先搜索或广度优先搜索，可以访问到 G 中的所有的顶点。在这两种搜索方法中，从一个已访问过的顶点 v_i 搜索到一个未曾访问过的邻接点 v_j，必定要经过 G 中的一条边(v_i, v_j)，由于两种方法对图中的 n 个顶点都仅访问一次，因此，除初始出发点外，对其余 $n-1$ 个顶点的访问将要经过 G 中的 $n-1$ 条边，这 $n-1$ 条边把 G 中的 n 个顶点连接成一个极小连通子图，从而得到 G 的一棵生成树。

由于可以利用图的遍历算法求得生成树，因此对算法 DFSM(或 DFSL)做调整就可得到深度优先生成树(DFS 生成树)的算法。分析算法可知，当 DFSM(i)递归调用 DFSM(j)时，v_i 是已访问过的顶点，v_j 是邻接于 v_i 的、未曾访问过且正待访问的顶点。因此，只需在 DFSM 算法的 if 语句中，在调用语句 DFSM(j)之前插入适当的语句，将边(v_i, v_j)输出或保存起来即可实现。

类似地，在算法 BFS(或 BFSL)中，若当前出队的元素是 v_i，待入队的元素是 v_j，则 v_i 是已访问过的顶点，v_j 是待访问而未曾访问过的、邻接于 v_i 的顶点。因而只需在 BFS 算法的 if 语句中插入适当语句，即可得到求广度优先生成树的算法。例如，从图 G_6 的顶点 v_1 出发所得到的 DFS 生成树和 BFS 生成树如图 7.21 所示。



(a) DFS生成树　　　　(b) BFS生成树

图 7.21　图 G_6 的 DFS 和 BFS 生成树

上面给出的生成树定义，是从连通图的观点出发、针对无向图而言的。此定义不仅仅适用于无向图，对有向图同样适用。若 G 是强连通的有向图，则从其中任一顶点 v 出发，都可以访问遍 G 中的所有顶点，从而得到以 v 为根的生成树。区别在于，有向图的深度优先生成树和广度优先生成树中的每一条边都是有向弧。

若 G 是非连通无向图，则要多次调用 DFS(或 BFS)算法，才能完成对 G 的遍历。因为每一次的外部调用，只能访问到 G 的一个连通分量的顶点集，这些顶点和遍历时所经过的边构成了该连通分量的一棵 DFS(或 BFS)生成树，而 G 的各个连通分量 DFS(或 BFS)生成树组成了 G 的 DFS(或 BFS)生成森林。类似地，若 G 是非强连通的有向图，且初始出发点又不是有向图的根，则遍历时一般也只能得到该有向图的生成森林。

连通图的生成树不是唯一的，因为从不同的顶点出发进行遍历，可以得到不同的生成树。如果连通图 G 是一个带权值的网图，则 G 的生成树的各边也是带权的。我们把生成树各边的权值总和称为生成树的权，并把权值最小的生成树称为 G 的最小生成树(Minimun Spanning Tree)。

生成树和最小生成树有许多重要的应用，求解生成树在许多领域中有重要意义。例如，电信实施工程师需要为一个镇的 6 个村庄架设通信网络做设计，v_1～v_6 表示村庄，之间的连线表示村与村之间的可通达的直线距离，如 v_1 到 v_2 是 16km，无连线表示有高山湖泊而无距离数据，如图 7.22 所示。那么，如何用最小的成本完成这次任务？

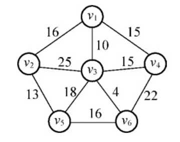


图 7.22　村庄间的连接图

由于在 n 个城市之间最多可设立的线路有 $n(n-1)/2$ 条，而把 n 个城市连接起来至少要有 $n-1$ 条线路，而且在每两个城市间设置一条线路，相应的都要付出一定的经济代价，所以，问题归结为：如何在 $n(n-1)/2$ 条线路中选择 $n-1$ 条，使得总耗费(代价)最小。

因此，可以用连通网来表示通信线路，顶点表示 n 个城市，边表示连接两个城市之间的线路，设置边的权值表示两个城市之间通信线路的长度或建造代价，则对 n 个顶点的连通网可以构造出许多不同的生成树，每棵生成树都可以是一个通信网，然后选择一棵总耗费最少的生成树(连通网的最小代价生成树)，使得建立的通信网络的线路的总长度最短或总代价最小，即要构造该图的一棵最小生成树。

构造最小生成树，要解决两个问题：尽可能选取权值小的边，但不能构成回路；选取 $n-1$ 条恰当的边以连接网的 n 个顶点。

7.4.2 最小生成树的构造

构造最小生成树可以有多种算法，其中大多数构造算法都是利用了最小生成树的 MST 性质：设 $G=(V, E)$是一个连通网络，U 是顶点集 V 的一个真子集。若(u, v)是 G 中所有的一个端点在 U 里(即 $u\in U$)、另一个端点不在 U 里(即 $v\in V-U$)的边中，具有最小权值的一条边，则一定存在 G 的一棵最小生成树包括此边(u, v)，该性质称为 MST 性质。

MST 性质可用反证法证明：假设 G 的任意一棵最小生成树中都不包含此边(u, v)。设 T 是 G 的一棵最小生成树，但不包含边(u, v)。由于 T 是树，且是连通的，因此有一条从 u 到 v 的路径；且该路径上必有一条连接两个顶点集 U 和 $V-U$ 的边(u', v')，其中 $u'\in U$，$v'\in V-U$，否则 u 和 v 不连通。当把边(u, v)加入树 T 时，得到一个含有边(u, v)的回路，如图 7.23 所示。删去边(u', v')，上述回路即被消除，由此得到一棵生成树 T'，T' 和 T 的区别仅在于用边(u, v)取代了 T 中的边(u', v')。因为(u, v)的权≤(u', v')的权，故 T' 的权≤T 的权，因此 T' 也是 G 的最小生成树，它包含边(u, v)，与假设矛盾。

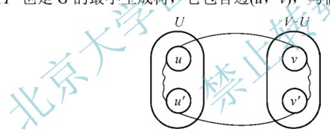


图 7.23 包含边(V, V-U)的回路

找连通图的最小生成树，经典的有两种算法 z 普里姆(Prim)算法和克鲁斯卡尔(Kruskal)算法。

1. 普里姆算法

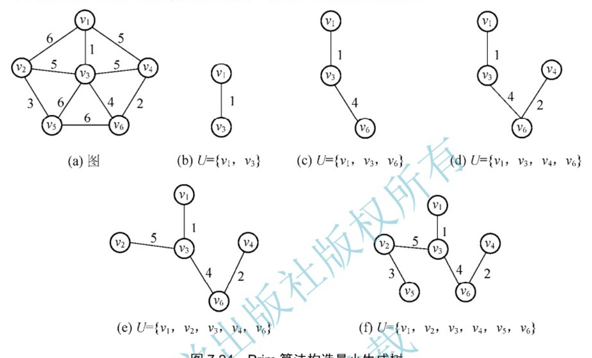
设 $G=(V, E)$是具有 n 个顶点的连通网，顶点集为 $V=\{v_1, v_2, \cdots, v_n\}$。设所求的最小生成树为 $T=(U, TE)$，其中 U 是最小生成树 T 中的顶点集，TE 是 T 中的边集，并将 G 中边上的权看作长度。

Prim 算法的基本思想：首先从 V 中任取一个顶点(以 v_1 为例)，将生成树 T 置为仅有一个节点 v_1 的树，即置入选顶点集 $U=\{v_1\}$，此时入选边集 $T=\{\}$，除 v_1 外的其他顶点构成待选顶点集 V-U；只要 U 是 V 的真子集，就在所有的其一个端点 v_i 已在 T(即 $v_i\in U$)、另一个端点 v_j 还未在 T(即 $v_j\in V-U$)的边中，找一条最短(即权最小)的边(v_i, v_j)，并把该条边(v_i, v_j)并入 T 的边集 TE，相应的顶点 v_j 加入入选顶点集 U。重复以上过程，每次往生成树里并入一个顶点和一条边，直至入选顶点集 U 包含了所有的顶点，即 $U=V$，而入选边集 TE

中有 $n-1$ 条边。MST 性质保证上述过程求得的 $T=(U, TE)$是 G 的一颗最小生成树。

显然，Prim 算法的关键是如何找到连接 U 和 $V-U$ 的最短边来扩充生成树 T。简单的方法就是在实施算法之前，将所有边进行排序，构造一个较小的候选边集合，且保证最短边属于该候选集，并且在每次找到一条最短边加入到入选边集 TE 时，调整待选边集合。

图 7.24 所示为用 Prim 算法找到的一棵最小生成树的过程。



(a) 图　(b) $U=\{v_1, v_3\}$　(c) $U=\{v_1, v_3, v_6\}$　(d) $U=\{v_1, v_3, v_4, v_6\}$

(e) $U=\{v_1, v_2, v_3, v_4, v_6\}$　(f) $U=\{v_1, v_2, v_3, v_4, v_5, v_6\}$

图 7.24　Prim 算法构造最小生成树

为实现这个算法，设连通图用邻接矩阵表示，对不存在的边，相应的矩阵元素用∞表示，实际编码可以去计算机允许的最大值存储。边的存储结构如下。

```
typedef struct{
    int end;            /*最短边的终点,起点是候选点自身*/
    int len;            /*边长,这里假设为整数*/
    }minedge[max+1];    /*max 是图中顶点数的最大值 */
```

Prim 算法的代码描述如下。

算法 7.7　Prim 算法。

```
void  Prim(Graph *g,int u)
 { int  v,k,j,min;
    for (v =1; v<=g->n; v++)
    if(v ! = u)
  {minedge[v].end=u;
   minedge[v].len = g->edges[v][u];
  }
minedge[u].len = 0;
for( k = 1; k< g->n;k++)
 {min = minedge[k].len;
    v = k;
```

```
for(j =1; j<g->n; j++)
if(minedge[j].len>0&&minedge[j].len<min)
 {min = minedge[j].len;
    v = j;
}
if(min=INTMAX)
 { printf ( "图不连通,无生成树!");
   return(0);
  }
printf("%d  %d",v, minedge[v].end);
minedge[v].len = -minedge[v].len;
for(j=1;j<=g->n;j++)
if(g-> edges [j][v]<minedge[j].len)
 {minedge[j].len = g-> edges [j][v];
    minedge[j].end = v;
  }
} }
```

由算法代码中的循环嵌套可知算法的时间复杂性是 $O(n^2)$，与边数无关，因此比较适合构造稠密图的最小生成树，即边数非常多的情况下使用。

2. 克鲁斯卡尔算法

设 $G=(V, E)$是连通网络，令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 $T=(V, \phi)$，T 中每个顶点自成一个连通分量。按照长度递增的顺序依次选择 E 中的边(u, v)，若该边的端点 u，v 分别是当前 T 的两个连通分量 T_1、T_2 中的顶点，则将该边加入到 T 中，T_1 和 T_2 也由此边连接成一个连通分量；若 u、v 是当前同一个连通分量中的顶点，则舍去此边(因为每个连通分量都是一棵树，此边添加到树中将形成回路)。以此类推，直到 T 中所有顶点都在同一连通分量上为止，T 便是 G 的一棵最小生成树。

对图 7.24(a)中的连通网络，按 Kruskal 算法构造最小生成树，其过程如图 7.25 所示。将边按长度进行排序(v_1, v_3)，(v_4, v_6)，(v_2, v_5)，(v_3, v_6)，(v_3, v_4)，(v_2, v_3)，(v_1, v_4)，(v_1, v_2)，(v_3, v_5)，(v_5, v_6)后，依次选取长度最短，且又都连通两个不同的连通分量的边(v_1, v_3)，(v_2, v_5)，(v_4, v_6)，将它们添加到 T 中，如图 7.24(a)～(c)所示；考虑当前最短边(v_3, v_6)，因为边的两端在不同的连通分量上，因此加入 T，如图 7.24(d)所示；选取边(v_3, v_4)，因为该边的两个端点在同一个连通分量上，若将其加入到 T 中，将会出现回路，故舍去这条边；同理，将边(v_2, v_3)加入到 T 中，舍去边(v_1, v_4)，便得到如图 7.25(e)所示的单个连通分量 T，它就是所求的一颗最小生成树。

实现这个算法要解决两个问题：采用什么样的存储结构，如何判断所选的边加入最小生成树后不产生回路。由于算法的每一步都是选择一条当前权值最小的且不会形成回路的边加入到最小生成树的边集中，因此中间过程一般会成森林。随着算法的进行，森林中的树将逐步连通(合并)成一棵树，即所求的最小生成树。因此，Kruskal 算法的基本思想是按边权递增的次序连通森林。

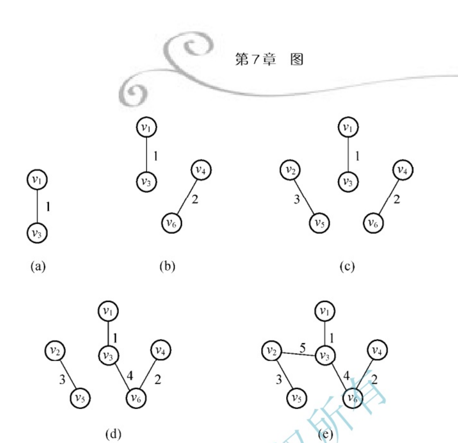


图 7.25　Kruskal 算法构造最小生成树

Kruskal 算法的粗略描述如下。

```
T=(V, φ)
while(T中所含边数<n-1)
{    从E中选取当前最短边(u,v);
     从E中删除边(u,v);
     if((u,v)并入T之后不产生回路)          将边(u,v)并入T;
}
```

克鲁斯卡尔算法代码描述如下。

算法 7.8　Kruskal 算法。

```
typedef  struct {
     int   v1,v2;
     int  len;
     }EdgeType;        /* 边的类型：两个端点号和边长*/
int parent[NMAX+1];  /*节点双亲的指针数组,设为全局量,NMAX为节点数最大值*/

int GetRoot(int  v)  /*查找节点v所在的树根,即查找双亲*/
 {  int   i;
        i = v;
while(parent[i]>0)i = parent[i];
return i;                          /*若无双亲(初始点),则双亲运算结果为其自己*/
    }

int  GetEdge(EdgeType  em [ ],int e) /*查找最短边在数组em中的编号,e为边数*/
 {int  i, j, min = max;                /* max最大的一个数*/
for (i =1 ; i<= e ; i++)
```

```
    if (em [i-1].len<min) { min = em [i-1].len;j=i-1;}
    return j;
    }
    void  Kruskal(EdqeType  em [ ],int n,int  e) /*n 为节点数,e 为边数*/
    {int  i,p1,p2,m,i0;
    for(i=1;i<= n;i++)/*初始节点为根,无双亲*/
    parent[i ] = -1;                    /*用于累计节点个数,此初值不能置为 0*/
    m = 1;
    while(m< n)      /*获取最短边,合并两棵树,共获取 n-1 条最短边得到一棵生成树*/
    {i0 = GetEdge(em,e);     /*获得最短边号,此号是所求边在数组 em 中的位置*/
      p1= GetRoot(em [i0].v1); /*获得最短边的两个顶点号,并求得两顶点所在树的根p1 和p2*/
      p2= GetRoot(em [i0].v2);
      if(p1= = p2)continue;          /*若连通分量相同,则不合并*/
      if( parent [p1] >parent [p2]  ) /*parent [p1] 与 parent [p2] 这时为负,因
根无双亲 */
       {parent [p2] = parent [p1] +parent [p2];
        parent[p1]= p2; }
    else {parent[ p1]= parent[p1]+parent[p2];      parent[p2]= p1;}
      printf (“%d%d%d\n”,m,em[i0].v1,em [i0].v2 ); /*输出第 m 条最短边*/
      m++;                      /*准备查找下一条最短边,共找 n-1 条*/
    }
    }
```

用 Kruskal 算法构造最小生成树的时间复杂度为 $O(e)$，与网中边的数目有关，因此适合于求稀疏图的最小生成树。

7.5 最 短 路 径

交通网络中往往需要解决如下的问题：两地之间是否有通路？如果有多条通路，则如何确定哪一条是最短的？如果将该交通网络用带权图来表示，则图中顶点表示城镇，边表示两个城镇之间的道路，边上权值表示两城镇间的距离、交通费用或途中所需的时间等信息。上述提出的问题就是在带权图(网)中求最短路径的问题，即两个顶点间长度最短的路径。“最短”的具体含义取决于边上权值所代表的意义。对于对非网图来说，所谓的最短路径指的是两顶点间经过的边数总和最小的路径，因为图中的边上无权值；而网图的最短路径，是指两顶点之间经过的边上权值之和最小的路径，并且称路径上的第一个顶点是源点(Source)，最后一个顶点为终点(Destnation)。

现实中两地间的路径往往是有向的，如 A 城到 B 城有一条公路，而且 A 城的海拔高于 B 城，若考虑到上坡和下坡的车速不同，则边<A，B>和边<B，A>上表示行驶时间的权值也将不同，也就是说<A，B>和<B，A>应该是两条不同的边。考虑到交通网络的这种有向性，本节将只讨论有向网的最短路径问题。为了方便讨论，设顶点集 $V=\{1，2，\cdots，n\}$，并假定所有边上的权值均为表示长度的非负实数。

7.5.1　单源最短路径

单源最短路径问题是指对于给定的有向网 G=(V，E)及单个源点 v，求从 v 到 G 的其余各个顶点的最短路径。

如果用图 7.26 所示的有向网 G_7 表示五个城市间的航线图，顶点表示不同的城市，弧上的权值表示运输费用，则求城市 1 到其他各城市的最小运输费用，实际上就是求 G_7 中顶点的最短路径问题。

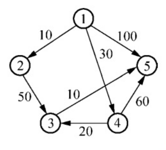


图 7.26　有向网 G_7

从有向网 G_7 可以看出，顶点 1 到其他各顶点的路径如下。

1 到 2 的路径有一条：1→2(10)，括号中的数是路径长度，即路径上的权值之和。

1 到 3 的路径有两条：1→2→3(60)，1→4→3(50)。

1 到 4 的路径有一条：1→4(30)。

1 到 5 的路径有四条：1→5(100)，1→4→5(90)，1→2→3→5(70)，1→4→3→5(60)。

分别选出 1 到其他各顶点的最短路径，并按路径长度递增顺序排列，得到：1→2(10)，1→4(30)，1→4→3(50)，1→4→3→5(60)。

观察上面的序列，可以发现规律：若按长度递增的次序生成从源点 v 到其他顶点的最短路径，则当前正在生成的最短路径上除终点以外，其余顶点的最短路径均已形成。将源点的最短路径看作已生成的源点到其自身的长度为 0 的路径。如图 7.26 中，若当前正在生成的是顶点 3 的最短路径，则该路径 1→4→3 上顶点 1 和 4 的最短路径在此以前已生成，因为它们的最短路径长度比顶点 3 的最短路径长度要小。

迪杰斯特拉(Dijkstra)算法正是基于上述规律而设计的，其算法使用了贪心(局部最优)策略，按路径长度递增的次序产生各顶点的最短路径，即利用局部最优来计算全局最优。算法的基本思想：设置两个顶点集 S 和 T，S 中存放已经确定的最短的路径的顶点，T 中存放待确定最短路径的顶点；初始状态时，S 中只有一个源点 v，T 中包含除源点外的其他顶点，而各顶点当前的最短路径长度为源点 v 到该顶点的弧上的权值，如果没有源点到该顶点的直接可达路径，则路径长度为无穷大；选取 T 中当前最短路径长度最小的一个顶点 v' 加入到 S 中，然后修改 T 中剩余顶点的当前最短路径长度，修改原则是当顶点 v'的最短路径长度与 v'到 T 中各顶点间的权值之和小于该顶点的当前最短路径长度时，用前者的值代替后者；重复上述过程，直到 S 中包含所有的顶点为止。

图 7.27 给出了有向网中的顶点 1 到其他各顶点的最短路径的过程，图中实线圈和虚线圈分别表示已确定和未确定最短路径的顶点，实线表示已确定了的最短路径上的弧，虚线表示有可达路径，但是未确定最短路径的弧。

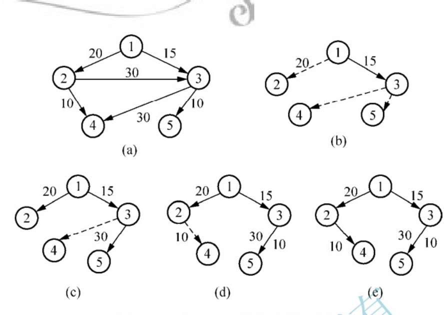
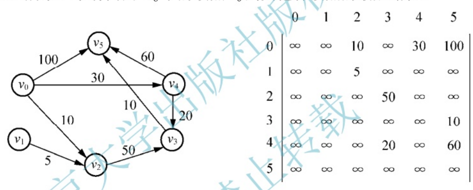


图 7.27　用 Dijkstra 算法求最短路径

下面用实例对该算法过程进行介绍。

如图 7.28 所示，求有向网 G_8 中的顶点 v_0 到其他各顶点的最短路径。



	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

图 7.28　有向网 G_8 和邻接矩阵

(1) 假设集合 V 是 G_8 的顶点集，S 是已找到最短路径的顶点集，T 是待确定最短路径的顶点集，则初始状态时，$S=\{v_0\}$，$T=\{v_1, v_2, v_3, v_4, v_5\}$。以后陆续将已求得的最短路径的顶点加入到 S 中，并在 T 中删除掉该顶点。当所有顶点都进入集合 S 时，则算法结束。用一维数组表示集合，如果顶点 v_i 加入到集合内，则对应数组值为 S[i]为 1，否则为 0。

(2) 将有向图用邻接矩阵表示，即用权值代替邻接矩阵中原来的 1，若顶点间无边则用∞表示。

(3) 用数组 Path[i]存放顶点最短路径上该点的前趋顶点，通过它可以找到路径上每个顶点的前趋，从而得到最短路径。

(4) 用一维数组 Disc[]存放各顶点的最短路径长度，则 Disc[i]表示源点 v_0 到某个顶点 v_i 的当前最短路径的长度，如果 v_0 到 v_i 无通路，则 Disc[i]=∞。每当有一个顶点进入集合 S 时，都要调整该数组中的最短路径值(在顶点 v_i 加入到 S 中前，相应的 Disc[i]都是中间结果)，当最后一个顶点进入集合 S 以后，修改完 Disc 中的值，即可得到源点到其他所有顶点的最短路径。

算法具体过程步骤如下。

(1) 初始，Disc[i]的值为 v_0 到 v_i 的弧的权值。

v_0 到顶点 v_2、v_4、v_5 的权值分别为 10，30，100，而到其他顶点没有可达通路，因此取最小值 10，即 Disc[2]=10 便是 v_0 到 v_2 的最短路径的长度，Path[2]=(v_0)。

(2) 寻找下一条最短路径。设下一条最短路径的终点是 v_j，则这条最短路径或者是(v_0，v_j)或者是 v_0 经过 v_2 到达 v_j 的路径。分析可知，v_0 和 v_5、v_4 直接连通，而且经过 v_2 可达 v_3，路径长度分别为 100、30、10+50=60，其中的最小值为 30，即 D[4]=30 便是 v_0 到 v_4 的最短路径的长度，Path[4]=(v_0)。

(3) 继续寻找下一条最短路径。设下一条最短路径的终点是 v_k，则这条最短路径或者是(v_0，v_k)、或者是 v_0 经过 v_2 或 v_4 到达 v_k 的路径。分析可得，v_0 直接可达 v_5，经过 v_2 可达 v_3，经过 v_4 可达 v_3 和 v_5，路径长度分别为 100、10+50=60、30+20=50、30+60=90，最小值为 50，即 D[3]=50 便是 v_0 到 v_3 的最短路径的长度，Path[3]=(v_4)。

以此类推，求出到所有顶点的最短路径。

迪杰斯特拉算法过程中，各顶点最短路径值的变化情况如表 7.1 所示。

表 7.1　Dijkstra 算法中各顶点路径变化

	v_0	v_1	v_2	v_3	v_4	v_5
S	1	0	0	0	0	0
Disc	0	∞	10	∞	30	100

	v_0	v_1	v_2	v_3	v_4	v_5
S	1	0	1	0	0	0
Disc	0	∞	10	60	30	100

	v_0	v_1	v_2	v_3	v_4	v_5
S	1	0	1	0	1	0
Disc	0	∞	10	30	30	90

	v_0	v_1	v_2	v_3	v_4	v_5
S	1	0	1	1	1	0
Disc	0	∞	10		30	60

	v_0	v_1	v_2	v_3	v_4	v_5
S	1	0	1	1	1	1
Disc	0	∞	10	30	30	60

每一次的路径长度调整都涉及两个操作：选择 S 为 0 的顶点中路径长度最短的那个顶点 v_i，将其 S 值设置为 1；然后调整 S 仍为 0 的所有顶点的当前最短路径长度。

最终得到的 v_0 到各顶点的最短路径及长度加表 7.2 所示。

表 7.2　有向网 G_8 中 v_0 到各顶点的最短路径和路径长度

源点	终点	最短路径	路径长度
v_0	v_1	无	∞
	v_2	$v_0 \to v_2$	10
	v_3	$v_0 \to v_4 \to v_3$	50
	v_4	$v_0 \to v_4$	30
	v_5	$v_0 \to v_4 \to v_3 \to v_5$	60

根据上述分析，假设用带权的邻接矩阵 arcs[*i*][*j*]来表示带权有向图，则迪杰斯特拉算法可描述如下。

(1) 初始，Disc[*i*]存放 v_0～v_i 各顶点的弧的权值，Disc[*i*]=arcs[0][*i*]，S={}。

(2) 重复执行 *n*-1 遍，每遍求出一条新的最短路径。

① 利用公式 D[j]=Min{D[i]|$v_i \in V\text{-}S$}得到一条新的从 v_0 出发的最短路径及新的终点 v_j，令 $S=S+\{v_j\}$。

② 利用 v_j 修改从 v_0 出发到集合 *V-S* 中任一顶点 v_k 可达路径的长度 ：D[j] + arcs[j][k]与 D[k]。

代码实现如下

算法 7.9　Dijkstra 算法。

```
void Dijkstra(Graph *g,int v,int *dist,int *path,char *s)
{
    int i,j,k ,pre ,min ;
    for(i=1;i<=g->n;i++)
    {
        s[i]= 0;//初始集合 s 为空
        dist[i] = g->adges[v][i];
        path[i] = V;
    }
    s[v] = 1;
    dist[v] = 0;
    for(i=1;i<g->n;i++)
     {
        min=INTMAX;
        for(j=1;j<g->n;j++)
        if (!s[j]&&dist[j]< min )
    {
        min=dist[j];
        k=j;
    }
    if(min = = INTMAX) break;
    s[k]=1; //将当前找到的到源点 v 距离最小的顶点 k 并入 s
    for(j=1;j<=g->n;j++)
    if(!s[j]&&dist[j]>dist[k]+g->adges[k][j])
```

```
            {
                dist[j]=dist[k]+g->adges[k][j];
                path[j]=k;   //顶点 j 的前趋是顶点 k
            }
        }
        for(i=1;i<= g->n;i++)
        {
            if(dist[i]= =INTMAX)
                {printf(" 无路径\n");continue;}
            printf("顶点%d 到源点的最短路径长:%d\n",i,dist [i] );
            pre=path[i];
            do
                 {
                    printf("<- %d", pre);
                    pre = path[pre];
                } while(pre! = v);
            }
        }
```

容易看出，算法 Dijkstra 的时间复杂度为 $O(n^2)$，占用的辅助空间是 $O(n)$。

7.5.2 所有顶点对间的最短路径

所有顶点对之间的最短路径问题是指对于给定的有向网 $G=(V, E)$，要给出的 G 中任意两个顶点 v_i、$v_j(v_i \neq v_j)$，找出 v_i 到 v_j 的最短路径。

解决此问题的一种方法是：依次把有向网络的每个顶点作为源点，重复执行迪杰斯特拉算法 n 次，即可求得每对顶点之间的最短路径。相当于在原有算法基础上，再增加一层循环，此时整个算法的时间复杂度是 $O(n^3)$。本小节要介绍另一个更为直接和简洁的解决这一问题的算法，它是由弗洛伊德(Floyd)提出的，其执行时间也是 $O(n^3)$。

Floyd 算法求各顶点间的最短路径长度的基本思想如下。

算法采用邻接矩阵 edges 存储带权有向网，有向网中的 n 个顶点从 v_0 开始编号，则顶点 v_i 到 v_j 的最短路径长度 edges[i][j]就是弧 $<v_i, v_j>$ 所对应的权值。当弧 $<v_i, v_j>$ 不存在时，edges[i][j]=∞；当 $v_i=v_j$ 时，edges[i][j]=0，即对角线上的元素为 0。

为了便于理解，先从直观上进行分析。对于 $0 \leqslant i, j \leqslant n-1$，若 v_i 到 v_j 有边，则从 v_i 到 v_j 存在一条长度为 edges[i][j]的路径。但它不一定是从 v_i 到 v_j 的最短路径，因为可能存在一条从 v_i 到 v_j，但包含其他顶点为中间点的路径。因此，应该依次考虑 v_i 到 v_j 能否有以顶点 v_0，v_1，…，v_{n-1} 为中间点的更短路径。

(1) 考虑从 v_i 到 v_j 是否有以顶点 v_0 为中间点的路径：v_i，v_0，v_j，即考虑 G 中是否有弧 $<v_i, v_0>$ 和 $<v_0, v_j>$，若有，则新路径 v_i、v_0、v_j 的长度是 edges[i][0]和 edges[0][j]之和，比较路径 v_i、v_j 和 v_i、v_0、v_j 的长度，并以较短者为当前所求得的最短路径。该路径是中间点序号不大于 0 的最短路径。

(2) 考虑从 v_i 到 v_j 是否有包含顶点 v_1 为中间的路径：v_i，…，v_1，…，v_j，若没有，则从 v_i 到 v_j 的当前最短路径仍然是步骤(1)中求出的，即从 v_i 到 v_j 的中间点序号不大于 0 的最

短路径；若有，则 v_i，…，v_1，…，v_j 可分解为两条路径 v_i，…，v_1 和 v_1，…，v_j，而这两条路径是前一次找到的中间点序号不大于 0 的最短路径，将这两条路径长度(已在前一次中求出)相加就得到路径 v_i，…，v_1，…，v_j 的长度，将该长度与前一次中求出的从 v_i 到 v_j 的中间点序号不大于 1 的最短路径长度做比较，取其较短者作为当前求得的从 v_i 到 v_j 的中间点序号不大于 1 的最短路径。

(3) 选择顶点 v_2 加入当前求得的从 v_i 到 v_j 中间点序号不大于 2 的最短路径中，按上述步骤进行比较，从未加入顶点 v_2 作为中间点的最短路径和加入顶点 v_2 作为中间点的新路径中选取较小者，作为当前求得的从 v_i 到 v_j 的中间点序号不大于 2 的最短路径。以此类推，直至考虑了顶点 v_{n-1} 加入当前从 v_i 到 v_j 的最短路径后，选出从 v_i 到 v_j 的中间点序号不大于 $n-1$ 的最短路径为止。由于 G 中顶点序号不大于 $n-1$，所以从 v_i 到 v_j 的中间点序号不大于 $n-1$ 的最短路径，已考虑了所有顶点作为中间点的可能性，因而它必然是从 v_i 到 v_j 的最短路径。

为了更直观地理解 Floyd 算法的原理，下面来分析一个最简单的三个顶点的连通网，如图 7.29 所示。

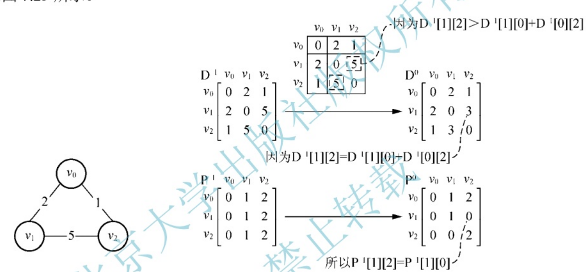


图 7.29　三个顶点连通网的 Floyd 算法示意图

定义两个二维数组 $D[3][3]$和 $P[3][3]$，$\boldsymbol{D}$ 表示顶点到顶点的最短路径权值和的矩阵，$\boldsymbol{P}$ 表示对应顶点的最小路径的前趋矩阵。初始状态时，我们将 $\boldsymbol{D}$ 命名为 $\boldsymbol{D}^{-1}$，它就是初始的图的邻接矩阵；将 $\boldsymbol{P}$ 命名为 $\boldsymbol{P}^{-1}$，初始化为图中所示的矩阵。

首先，分析所有的顶点经过 v_0 后到达另一个顶点的最短路径。因为只有三个顶点，因此需要查看 $v_1 \to v_0 \to v_2$，得到 $\boldsymbol{D}^{-1}[1][0]+\boldsymbol{D}^{-1}[0][2]=2+1=3$。$\boldsymbol{D}^{-1}[1][2]$表示的是 $v_1 \to v_2$ 的权值为 5，可以发现，$\boldsymbol{D}^{-1}[1][2]> \boldsymbol{D}^{-1}[1][0]+\boldsymbol{D}^{-1}[0][2]$，也就是说 $v_1 \to v_0 \to v_2$ 比直接 $v_1 \to v_2$ 的距离要短。所以将 $\boldsymbol{D}^{-1}[1][2]= \boldsymbol{D}^{-1}[1][0]+\boldsymbol{D}^{-1}[0][2]=3$，同样的，$\boldsymbol{D}^{-1}[2][1]=3$，于是有了增加顶点 v_0 的调整后的 $\boldsymbol{D}^0$ 矩阵。因为有了变化，所以 $\boldsymbol{P}$ 矩阵对应的 $\boldsymbol{P}^{-1}[1][2]$和 $\boldsymbol{P}^{-1}[2][1]$也修改为当前中转顶点 v_0 的下标 0，于是就有了 $\boldsymbol{P}^0$。

然后，在 $\boldsymbol{D}^0$ 和 $\boldsymbol{P}^0$ 的基础上继续处理所有顶点经过 v_1 和 v_2 后到达另一个顶点的最短路径，得到 $\boldsymbol{D}^1$ 和 $\boldsymbol{P}^1$、$\boldsymbol{D}^2$ 和 $\boldsymbol{P}^2$，完成所有顶点到顶点的最短路径的计算。

分析上述实例的算法实施过程，可以发现实现上述算法的关键在于要保留每一步所求得的、所有顶点对之间的当前最短路径长度，设 n 个顶点从 1 开始编号，为此需要定义一个 $n \times n$ 的方阵序列 $\boldsymbol{A}^0$，$\boldsymbol{A}^1$，…，$\boldsymbol{A}^n$，来保存当前求得的所有顶点对之间的最短路径长度。其中，$\boldsymbol{A}^k[i][j]$表示从 v_i 到 v_j 的、中间点序号不大于 k 的最短路径长度($0 \leqslant k \leqslant n-1$)。

特别地，A^0等于G的邻接矩阵*edges*，$A^0[i][j]$(即edges$[i][j]$)表示从v_i到v_j不经过任何中间顶点的最短路径长度，$A^n[i][j]$就是从v_i到v_j的最短路径长度($0 \leqslant k \leqslant n$)。

Floyd算法的基本策略是从A^0开始，递推地生成矩阵序列A^1，A^2，…，A^n，所以如何由已求得的矩阵A^{k-1}推出A^k是关键。对于任何顶点对v_i、v_j，从顶点v_i到顶点v_j的、中间点序号不大于k的最短路径只有两种情况：一种是中间不经过顶点k，那么它仍然是前一次求出的从v_i到v_j的、中间点序号不大于k−1的最短路径，$A^k[i][j]=A^{k-1}[i][j]$；另一种是中间经过顶点k，该路径由两段路径i，…，k和k，…，j组成，它们都是前一次已求出的中间点序号不大于k−1的最短路径，其长度分别为$A_{k-1}[i][k]$和$A_{k-1}[k][j]$，因此，有$A_k[i][k]=A^{k-1}[i][k]+A^{k-1}[k][j]$，于是，可得到$A^k[i][j]$的递推公式：

$A^0[i][j]=\text{edges}[i][j]$

$A^{k+1}[i][j]=\min\{A^k[i][j],\ A^k[i][k]+A^k[k][j]\} \qquad 0 \leqslant k \leqslant n-1;\ 0 \leqslant i,\ j < n$

另外，为了得到最短路径本身，还必须设置一个路径矩阵path$[n][n]$，它也是迭代产生的，$\text{path}^{k+1}[i][j]$，是从i到j的中间点序号不大于k+1的最短路径上顶点i的后继顶点。算法结束时，由path$[i][j]$的值可以得到从i到j的最短路径上各个顶点。

初始时，path$[i][j]$=0表示从v_i到v_j的路径是直达的，中间不经过其他的顶点。以后，当考虑让路径经过某个顶点k时，若使路径更短，则在修改$A^k[i][j]$时，要令$\text{path}^k[i][j]=k$。如果想要输出最短路径顶点序列，则只需用一个循环读取即可实现，因为所有最短路径的顶点信息都包含在矩阵path中。设$\text{path}^n[i][j]=k$，即从v_i到v_j的最短路径经过顶点k，该路径上还有哪些顶点呢？只要去查找$\text{path}^n[i][k]$和$\text{path}^n[k][j]$即可，以此类推，直到所查元素为0为止。

弗洛伊德Floyd算法代码实现如下、

算法7.10 Floyd算法。


```
void  Floyd(Graph *g)
 {
  int i,j,k,next;
  for( i=1;i<=g->n;i++)
   for(j=1;j<=g->n;j++)
 {
    A[i][j] = g->edges[i][j];
    path[i][j]= 0;
   }
   for(k=1;k<=g->n;k++)
    for( i=1;i<=g->n ;i++)
      for(j=1;j<=g->n;j++)
       if(A[i][k]+A[k][j]< A[i][j])
        { A[i][j]=A[i][k]+A[k][j];
          path[i][j]= k;
        }
for(i=1;i<=g->n;i++)
   for(j= 1;j<=g->n;j++)
 { if (A[i][j] = = INTMAX)
        { printf("无路径\n");continue;}
    printf("顶点%d路径长度%d", i,A[i][j]);
      next =path[i][j];
      printf("路径为: %d",i );
```

```
        do {
            printf(" ->%d", next);
            next=path[next][j] ;
           } while(next! = j);
        }
    }
```

以有向网络 G_9 为例实施上述算法，迭代过程中 $\boldsymbol{A}$ 和 path 的变化及其最终结果如图 7.30 所示。

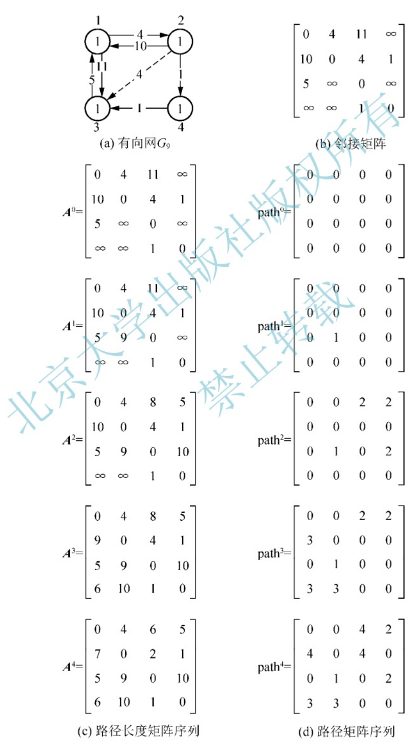


图 7.30 用弗洛伊德算法求各顶点间的最短路径

从上面例子得出的结果如何求顶点 4 到顶点 2 所经过的最短路径的顶点序列呢？先从 $path^4[4][2]=3$ 开始，说明由顶点 4 到顶点 2 先经过顶点 3，即路径上顶点 4 的后继顶点是顶点 3，$path^4[4][2]=<path^4[4][3], 3, path^4[3][2]>$，然后分析 $path^4[4][3]$和 $path^4[3][2]$，由于 $path^4[4][3]=0$，于是舍去 $path^4[4][3]$，而 $path^4[3][2]=1$，于是由顶点 4 到顶点 2 经过的第二个顶点是 1，即 $path^4[4][2]=<4, 3, path^4[3][1], path^4[1][2]>$，再分析 $path^4[3][1]$和 $path^4[1][2]$，因为两个的值都等于 0，因此顶点搜寻完毕，最后求的顶点 4 到顶点 2 的路径为(4→3→1→2)，且路径的长度(最短)是 $A^4[4][2]=10$。

7.6　校园电子导航平台的实现

(1) 用邻接矩阵保存各景点信息及景点间路径的信息。

```
typedef struct ArcCell{
    int adj;                    // 相邻景点之间的路程
    char *info;
}ArcCell;                       // 定义边的类型

typedef struct VertexType{
    int number;                 // 景点编号
    char *sight;                // 景点名称
    char *description;          // 景点描述
}VertexType;                    // 定义顶点的类型

typedef struct{
    VertexType vex[NUM];        // 图中的顶点即为景点
    ArcCell arcs[NUM][NUM];     // 图中的边即为景点间的距离
    int vexnum,arcnum;          // 顶点数,边数
}MGraph;                        // 定义图的类型
```


(2) 主程序代码如下。

```
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#define Max 32767
#define NUM 11

MGraph G;                           // 把图定义为全局变量
int P[NUM][NUM];                    // 路径存在标志
long int D[NUM];                    // 辅助变量存储最短路径长度

void CreateUDN(int v,int a);        // 创建图的函数
void pingmu();                      // 屏幕输出函数
```

```
void introduce();                          //景点介绍函数
void ShortestPath(int num);                //用 Dijkstra 算法求最短路径函数
void output(int sight1,int sight2);        //输出函数
void PrintMGraph();                        //输出显示邻接矩阵信息
char Menu();                               // 主菜单
void search();                             // 查询景点信息
char SearchMenu();                         // 查询子菜单

int main()                                 // 主函数
{
    int v0,v1;
    char ck;
    CreateUDN(NUM,11);
    do
    {
        ck=Menu();
        switch(ck)
        {
            case'1':
                introduce();               //景点介绍函数
                printf("\n\n%-25s\n\n",G.vex[0].description);
                getchar();
                getchar();
                break;
            case '2':
                system("cls");
                pingmu();
                printf("\n\n 请选择起点景点(1～10): ");
                scanf("%d",&v0);
                printf("请选择终点景点(1～10): ");
                scanf("%d",&v1);
                ShortestPath(v0);          // 计算两个景点之间的最短路径
                output(v0,v1);             // 输出结果
                printf("\n\n 请按回车键继续...\n");
                getchar();
                getchar();
                break;
            case '3':
                search();
                break;
            case'4':
                PrintMGraph();
                printf("\n\n 请按回车键继续...\n");
```

```
                getchar();
                getchar();
                break;

        }
    }while(ck!='e');
    return 0;
}

char Menu()       // 主菜单显示
{
    char c;
    int flag;
    do{
    flag=1;
    system("cls");
    pingmu();
    introduce();
    printf("\n ┌──────────────────────────────┐ \n");
    printf(" │                              │ \n");
    printf(" │  1.学校简介                  │ \n");
    printf(" │  2.查询景点路径              │ \n");
    printf(" │  3.查询景点信息              │ \n");
    printf(" │  4.查询各景点之间的距离      │ \n");
    printf(" │  e.退出                      │ \n");
    printf(" │                              │ \n");
    printf(" └──────────────────────────────┘ \n");
    printf("请输入您的选择：");
    scanf("%c",&c);
    if(c=='1'||c=='2'||c=='3'||c=='4'||c=='5'||c=='e')
    flag=0;
    }while(flag);
    return c;
}

char SearchMenu() // 查询子菜单
{
    char c;
    int flag;
    do{
    flag=1;
    system("cls");
    pingmu();
```

```
    introduce();
    printf("\n ┌──────────────────────────────┐ \n");
    printf(" │                              │ \n");
    printf(" │  1.按照景点编号查询          │ \n");
    printf(" │  2.按照景点名称查询          │ \n");
    printf(" │  e.返回                      │ \n");
    printf(" │                              │ \n");
    printf(" └──────────────────────────────┘ \n");
    printf("\t 请输入您的选择: ");
    scanf("%c",&c);
    if(c=='1'||c=='2'||c=='e')
    flag=0;
    }while(flag);
    return c;
}

void search() // 查询景点信息
{
    int num;
    int i;
    char c;
    char name[20];
    do
    {
        system("cls");
        c=SearchMenu();
        switch (c)
        {
            case '1':
                system("cls");
                introduce();
                pingmu();
                printf("\n\n 请输入您要查找的景点编号: ");
                scanf("%d",&num);
                for(i=0;i<NUM;i++)
                {
                 if(num==G.vex[i].number)
                 {
                    printf("\n\n 您要查找景点信息如下:");
                    printf("\n\n%-25s\n\n",G.vex[i].description);
                    printf("\n 按回车键返回...");
                    getchar();
                    getchar();
```

```
                    break;
                }
            }
            if(i==NUM)
            {
                printf("\n\n没有找到！");
                printf("\n\n按回车键返回...");
                getchar();
                getchar();
            }
            break;
        case '2':
            system("cls");
            pingmu();
            introduce();
            printf("\n\n请输入您要查找的景点名称：");
            scanf("%s",name);
            for(i=1;i<NUM;i++)
            {
                if(!strcmp(name,G.vex[i].sight))
                {
                    printf("\n\n您要查找景点信息如下:");
                    printf("\n\n%-25s\n\n",G.vex[i].description);
                    printf("\n按回车键返回...");
                    getchar();
                    getchar();
                    break;
                }
            }
            if(i==NUM)
            {
                printf("\n\n没有找到！");
                printf("\n\n按回车键返回...");
                getchar();
                getchar();
            }
            break;
        }
    }while(c!='e');
}


void CreateUDN(int v,int a)  // 创建图的函数
```

```
    {
        int i,j;
        G.vexnum=v;                    // 初始化结构中的景点数和边数
        G.arcnum=a;
        for(i=1;i<G.vexnum;++i)
            G.vex[i].number=i;         // 初始化每一个景点的编号
            // 初始化每一个景点名称及其景点描述
        G.vex[0].sight="学校简介";
        G.vex[1].sight="南大门";
        G.vex[2].sight="行政中心";
        G.vex[3].sight="树人礼堂";
        G.vex[4].sight="B2 楼";
        G.vex[5].sight="田径场";
        G.vex[6].sight="致勤楼";
        G.vex[7].sight="树人之家";
        G.vex[8].sight="教学楼";
        G.vex[9].sight="北校门";
        G.vex[10].sight="图书信息中心";
        // 这里把所有的边假定为 32767,含义是这两个景点之间不可到达
        for(i=1;i<G.vexnum;++i)
        {
            for(j=1;j<G.vexnum;++j)
            {
                G.arcs[i][j].adj=Max;
                G.arcs[i][j].info=NULL;
            }
        }
    /*下面是可直接到达的景点间的距离,由于两个景点间距离是互相的,所以要对图中对称的边同
时赋值*/
        G.arcs[1][8].adj=G.arcs[8][1].adj=50;
        G.arcs[1][10].adj=G.arcs[10][1].adj=100;
        G.arcs[2][8].adj=G.arcs[8][2].adj=150;
        G.arcs[2][9].adj=G.arcs[9][2].adj=20;
        G.arcs[10][3].adj=G.arcs[3][10].adj=30;
        G.arcs[9][5].adj=G.arcs[5][9].adj=200;
        G.arcs[1][4].adj=G.arcs[4][1].adj=70;
        G.arcs[5][6].adj=G.arcs[6][5].adj=50;
        G.arcs[10][7].adj=G.arcs[7][10].adj=150;
        G.arcs[2][3].adj=G.arcs[3][2].adj=50;
        G.arcs[3][7].adj=G.arcs[7][3].adj=100;
        G.arcs[4][10].adj=G.arcs[10][4].adj=20;
        G.arcs[4][3].adj=G.arcs[3][4].adj=50;
        G.arcs[10][2].adj=G.arcs[2][10].adj=50;
```

```
        G.arcs[8][10].adj=G.arcs[10][8].adj=20;
    }

    void PrintMGraph()     // 输出邻接矩阵
    {
        int i,j;
        printf("\n=================================================\n\n");
        for(i=1;i<G.vexnum;++i)
        {
            printf("%s ",G.vex[i].sight);
        }
        printf("\n");
        for(i=1;i<G.vexnum;++i)
        {
            printf("\n\n%s ",G.vex[i].sight);
            for(j=1;j<G.vexnum;++j)
            {
                if(G.arcs[i][j].adj==Max)
                    printf("%5c",'@');
                else
                    printf(" %5d",G.arcs[i][j].adj);
            }
        }
        printf("\n\n\n\n==============================================\n\n\n");
    }

    void introduce() // 景点介绍函数
    {
        int i;
        for(i=1;i<=NUM;i++)
        {
        G.vex[0].description="浙江树人学院创办于 1984 年,是改革开放以来我国最早成
立并经国家教育部批准的首批全日制民办普通高校之一。2003 年,学校升为本科院校。";
        G.vex[1].description="学校的正大门,位于树人街上";
        G.vex[2].description="学校的各行政机构所在地,包括财务处、科研处、人事处等";
        G.vex[3].description="学校的大礼堂,承担全院大会,周末还有电影放映";
        G.vex[4].description="学院的办公楼,老师们都在这里办公";
        G.vex[5].description="运动的地方,早上升国旗的地方,每年的运动会都在这里召开";
        G.vex[6].description="学校最高的宿舍楼,只是电梯才两个,异常拥挤";
        G.vex[7].description="食堂,同学们吃饭的地方,菜的种类很多,";
        G.vex[8].description="4 大教学楼 A1-A4,大部分课程都安排在这里,痛并快乐着的
地方";
        G.vex[9].description="学校的后校门,从这里出去,就是杭城著名的舟山东路小吃
```

```
街,小吃的味道很不错";
            G.vex[10].description="图书馆里环境非常好,书籍很多,安静又有学习氛围,很多
同学喜欢在这里自习";
            }
        }

        void pingmu()  // 屏幕输出函数
        {
            int i;
            printf("\n%c%c%c%c%c%c%c%c%c%c%c 欢迎来到浙江树人大学%c%c%c%c%c%
c%c%c%c\n\n",6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6);
            printf("%c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c
%c %c\n",3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3);
            printf("%c\t\t 学校概况\t\t%c\n",6,6);
            printf("%c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c %c
%c %c\n",3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3);
            for(i=1;i<NUM;i++)
            {
                printf("%c\t\t(%2d)%-20s%c\t\t\t\n",1,i,G.vex[i].sight,1);  //
输出景点列表
            }
            printf("%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%
c%c%c%c%c%c%c%c%c%c%c%c\n\n",1,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6
,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,1);
        }

        void ShortestPath(int num)   // Dijkstra 算法最短路径函数,num 为入口点的编号
        {
            int v,w,i,t;                    // i、w 和 v 为计数变量
            int final[NUM];
            int min;
            for(v=1;v<NUM;v++)
            {
                final[v]=0;                 // 假设从顶点 num 到顶点 v 没有最短路径
                D[v]=G.arcs[num][v].adj;// 将与之相关的权值放入 D 中存放
                for(w=1;w<NUM;w++)          // 设置为空路径
                    P[v][w]=0;
                if(D[v]<32767)              // 存在路径
                {
                    P[v][num]=1;            // 将存在标志置为 1
                    P[v][v]=1;              // 自身到自身
                }
            }
```

```
        D[num]=0;
        final[num]=1;          // 初始化 num 顶点属于 S 集合
    // 开始主循环,每一次求得 num 到某个顶点的最短路径,并将其加入到 S 集合中
        for(i=1;i<NUM;++i)         // 其余 G.vexnum-1 个顶点
        {
            min=Max;               // 当前所知离顶点 num 的最近距离
            for(w=1;w<NUM;++w)
                if(!final[w])      // w 顶点在 V-S 中
                    if(D[w]<min) // w 顶点离 num 顶点更近
                    {
                        v=w;
                        min=D[w];
                    }
            final[v]=1;            // 离 num 顶点更近的 v 加入到 S 集合中
            for(w=1;w<NUM;++w)     // 更新当前最短路径及其距离
               if(!final[w]&&((min+G.arcs[v][w].adj)<D[w]))/*若不在S集合中,
并且比以前所查找到的路径都短就更新当前路径*/
                {
                    D[w]=min+G.arcs[v][w].adj;
                    for(t=0;t<NUM;t++)
                     P[w][t]=P[v][t];
                     P[w][w]=1;
                }
        }
    }

    void output(int sight1,int sight2)   // 输出函数
    {
        int a,b,c,d,q=0;
        a=sight2;                            // 将景点二赋值给 a
        if(a!=sight1)          // 如果景点二不和景点一输入重合,则进行如下操作
        {
            printf("\n 从%s 到%s 的最短路径是",G.vex[sight1].sight,G.vex
[sight2].sight);/*输出提示信息*/
            printf("(最短距离为 %dm.)\n\n\t",D[a]);/*输出 sight1 到 sight2 的最
短路径长度,存放在 D[]数组中*/
            printf("%s",G.vex[sight1].sight);    // 输出景点一的名称
            d=sight1;                            // 将景点一的编号赋值给 d
            for(c=0;c<NUM;++c)
            {
                gate:; // 标号,可以作为 goto 语句跳转的位置
                P[a][sight1]=0;
                for(b=0;b<NUM;b++)
```

```
                {
                    if(G.arcs[d][b].adj<32767&&P[a][b]) /*如果景点一和它的一
个临界点之间存在路径且为最短路径,则进行如下操作*/
                        {
                        printf("-->%s",G.vex[b].sight); // 输出此节点的名称
                        q=q+1; // 计数变量加一,满 8 控制输出时换行
                        P[a][b]=0;
                        d=b; // 将 b 作为出发点进行下一次循环输出,如此反复
                        if(q%8==0) printf("\n");
                            goto gate;
                    }
                }
            }
        }
    }
```

独立实践

请读者在上述功能的基础上增加“景点的遍历游览”功能，即设计路线从任一景点出发，游历完所有的景点。

小　　结

图是一种复杂的非线性的数据结构，具有广泛的应用背景，在其讲解中，涉及了数组、链表、栈、队列、树等之前学的几乎所有数据结构，因此学好图，基本等于理解了数据结构这门课的精髓。

本章介绍了图的基本概念和两种常用的存储结构——邻接矩阵和邻接表，用什么存储结构需要具体问题具体分析，通常稠密图或读写数据较多、结构修改较少的图适合用邻接矩阵存储，反之应该考虑邻接表。图的遍历是图的一种主要操作，分为深度和广度两种。

关于图的应用，本章介绍了最小生成树和最短路径问题。前者的 Prim 算法是走一步看一步的思维方式，逐步生成最小生成树；而 Kruskal 算法则更有全局意识，直接从图中最短权值边入手，寻找最终答案。最短路径问题在现实中应用非常多，Dijkstra 算法和 Floyd 算法是常用的算法。

习　　题

一、填空题

1．图有________、________等存储结构，遍历图有________、________等方法。

2．有向图 G 用邻接表矩阵存储，其第 i 行的所有元素之和等于顶点 i 的________。

3．用 Prim 算法求具有 n 个顶点 e 条边的图的最小生成树的时间复杂度为________；用 Kruskal 算法计算的时间复杂度是________。

4．图的深度优先遍历序列________唯一的。

5．用 Dijkstra 算法求某一顶点到其余各顶点间的最短路径是按路径长度________的次序来得到最短路径的。

6．图的生成树是________(请填“极大”或“极小”)连通子图。

7．具有 n 个顶点的无向完全图的边的总数为________条；而具有 n 个顶点的有向完全图的边的总数为________条。

8．DFS 和 BFS 遍历分别采用________和________的数据结构来存储顶点，当要求连通图的生成树的高度最小时，应采用的遍历方法是________。

9．一个连通图的生成树是一个________连通子图，n 个顶点的生成树有________条边。

二、判断题

1．图可以没有边，但不能没有顶点。(　　)

2．带权图的最小生成树是唯一的。(　　)

3．有向图中的一个顶点的度是该顶点的出度。(　　)

4．对连通图进行深度优先遍历可以访问到该图中的所有顶点。(　　)

5．一个图的广度优先搜索树是唯一的。(　　)

6．求最小生成树时，Prim 算法在边较少、节点较多时效率较高。(　　)

7．在 n 个节点的无向图中，若边数大于 n-1，则该图必是连通图。(　　)

8．邻接矩阵存储一个图时，假设图的顶点个数为 n，则为邻接矩阵所付出的存储空间就是 n^2 个(具体大小是单个元素所占字节数)，而与边数无关。(　　)

9．邻接表法只用于有向图的存储，邻接矩阵对于有向图和无向图的存储都适用。(　　)

三、选择题

1．在一个图中，所有顶点的度数之和等于图的边数的(　　)倍。

A．1/2　　B．1　　C．2　　D．4

2．在一个有向图中，所有顶点的入度之和等于所有顶点的出度之和的(　　)倍。

A．1/2　　B．1　　C．2　　D．4

3．有 8 个节点的无向图最多有(　　)条边。

A．14　　B．28　　C．56　　D．112

4．有 8 个节点的无向连通图最少有(　　)条边。

A．5　　B．6　　C．7　　D．8

5．用邻接表表示图进行深度优先遍历时，通常是采用 (　　) 来实现算法的。

A．栈　　B．队列　　C．树　　D．图

6．已知图的邻接表如图 7.31 所示，根据算法，则从顶点 0 出发按深度优先遍历的节点序列是(　　)。

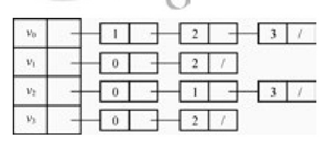


图 7.31　某图的邻接表

A．0 1 3 2　　B．0 2 3 1　　C．0 3 2 1　　D．0 1 2 3

7．深度优先遍历类似于二叉树的(　　)。

A．先根遍历　　B．中根遍历　　C．后根遍历　　D．层次遍历

8．广度优先遍历类似于二叉树的(　　)。

A．先根遍历　　B．中根遍历　　C．后根遍历　　D．层次遍历

9．已知图的邻接矩阵如图 7.32 所示，根据算法思想，则从顶点 0 出发按深度优先遍历的节点序列是(　　)。

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

图 7.32　某图的邻接矩阵

A．0 2 4 3 1 5 6　　B．0 1 3 5 6 4 2　　C．0 4 2 3 1 6 5　　D．0 1 3 4 2 5 6

10．用邻接表表示图进行广度优先遍历时，通常是采用(　　)　来实现算法的。

A．栈　　B．队列　　C．树　　D．图

四、简答题

1．已知如图 7.33 所示的有向图，请给出该图的每个顶点的入/出度，邻接矩阵，邻接表，逆邻接表。

2．已知如图 7.34 所示的无向带权图，请

(1) 写出它的邻接矩阵，并按 Prim 算法求其最小生成树。

(2) 写出它的邻接表，并按 Kruskal 算法求其最小生成树。

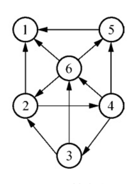


图 7.33　某有向图

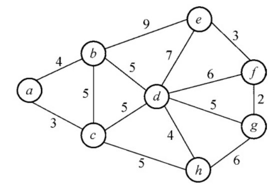


图 7.34　无向带权图

3．已知二维数组表示的图的邻接矩阵如图 7.35 所示。试分别画出自顶点 a 出发进行遍历所得的深度优先生成树和广度优先生成树。

4．试利用 Dijkstra 算法求图 7.36 所示的有向图 GA 中从顶点 a 到其他各顶点间的最短路径。

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	1	0	1	0
2	0	0	1	0	0	0	1	0	0	0
3	0	0	0	1	0	0	0	1	0	0
4	0	0	0	0	1	0	0	0	1	0
5	0	0	0	0	0	1	0	0	0	1
6	1	1	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0	1
8	1	0	0	1	0	0	0	0	1	0
9	0	0	0	0	1	0	1	0	0	1
10	1	0	0	0	0	1	0	0	0	0

图 7.35 某图的邻接矩阵

图 7.36 有向图 GA

5．给定下列网 G，则

(1) 试找出网 G 的最小生成树，画出其逻辑结构图。

(2) 用两种不同的表示法画出网 G 的存储结构图(见图 7.37)。

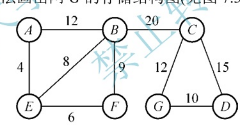


图 7.37 存储结构图

五、编程题

1．设计算法求解距离顶点 v_0 最远的一个顶点。

2．设有向图以邻接表存储，设计算法删除图中的弧<v_i，v_j>。

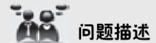
3．设计算法判断无向图 G 是否连通。若连通则返回 TRUE，否则返回 FALSE。

4．写出图的深度优先搜索算法的非递归算法。

5．设有向图 G 有 n 个顶点(用 1，2，3，…，n)表示，e 条边，编写一个算法根据其邻接表生成其逆邻接表，要求算法复杂度为 $O(n+e)$。



排　　序



问题描述

奥运会奖牌排名系统

在奥运会的若干单项比赛中，有多支参赛团体，每项比赛设金、银、铜牌，无并列奖项，如表 8.1 所示。请实现以下功能：输出奖牌榜，基本规则为金牌数多者名次优先，金牌数相等则比较银牌数，银牌数相同则比较铜牌数，如全部相等则为并列名次，并按表格形式输出；对奖牌总数进行排名，否则为并列名次，并按表格形式输出。

表 8.1　奥运会奖牌提成名示例

国家名	英文缩写	金牌数	银牌数	铜牌数	总数
中国	CHN	51	21	28	100
俄罗斯	RUS	23	21	28	72
美国	USA	36	38	36	110
……	…	…	…	…	…

8.1　概　　述

排序(Sorting)是数据处理中经常使用的一种重要运算。如何进行排序，特别是高效地进行排序是计算机应用中的重要课题之一。

在本章中，我们假定被排序的对象是由一组记录组成的文件，而记录则由若干个数据项(或域)组成，其中有一项可用来标识一个记录，称为关键字项，该数据项的值称为关键字。关键字可用来作为排序运算的依据，它可以是数学类型，也可以是字符类型，选取记录中的哪一项作为关键字，根据问题的要求而定。例如，在高考成绩统计中将每个考生作为一个记录，它包含准考证号，姓名、语文、外语、物理、化学、生物的分数和总分数等内容，若要唯一地标识一个考生的记录，则必须用“准考证号”作为关键字；若要按照考生的总分数排名次，则需要用“总分数”作为关键字。

所谓排序，是把一组记录或数据元素的无序序列按照某个关键字值(关键字)递增或递减的次序重新排列的过程。若给定的文件含有 n 个记录$\{R_1, R_2, \cdots, R_n\}$，它们的关键字分别是$\{k_1, k_2, \cdots, k_n\}$，需确定 1，2，…，$n$ 的一种排序 i_1，i_2，…，i_n，使其相应的关键字满足如下关系：$k_{i1} \leqslant k_{i2} \leqslant \cdots \leqslant k_{in}$，即使得$\{R_1, R_2, \cdots, R_n\}$的序列成为一个按关键字有

序的序列，也就是$\{R_{i1}, R_{i2}, \cdots, R_{in}\}$。这个将原有表中任意顺序的记录变成一个按关键字有序排列的过程称为排序。

待排序记录序列可以用顺序存储结构或链式存储结构表示。在本章讨论中，若无特别说明，均假定待排序记录序列采用顺序表结构来存储，即用一位数组实现，并且假定按关键字非递减方式排序。为了简单起见，假设关键字类型是整型，此结构定义适用于下面讲的所有的排序算法，其定义如下。

```
    typedef int KeyType;
    typedef struct{
        KeyType key;                  //关键字域
        InfoType otherinfo;           //记录的其他域,根据具体应用来定义
    }RecType;                         //记录类型
    typedef RecType SqList[n+1];      //记录类型的数组,n 为文件记录总数
```

排序算法的稳定性：当待排序记录的关键字均不相同时，则排序的结果是唯一的，否则排序的结果不一定唯一。如果待排序的文件中，存在有多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，则称这种排序方法是稳定的；反之，若具有相同关键字的记录之间的相对次序发生变化，则称这种排序方法是不稳定的。

各种排序方法可以按照不同的原则加以分类，在排序过程中，若整个文件都放在内存中处理，排序时不涉及数据内、外存交换，则称之为内部排序(简称内排序)；反之，若排序过程中要进行数据的内、外存交换，则称之为外部排序(简称外排序)。内排序适用于记录个数不是很多的小文件，外排序则适用于记录个数太多，不能一次将其全部记录放入内存的大文件。本章只讨论常用的内排序方法。按所用的策略不同，内排序方法可以分为五类 ：插入排序、选择排序、交换排序、归并排序和分配排序。

要在繁多的排序方法中，简单地判断哪一种算法好，以便能普遍选用是困难的。评论排序算法好坏的标准主要有两条：第一条是算法执行时所需的时间，第二条是执行算法所需要的辅助存储空间。另外，算法本身的复杂程度也是考虑的一个因素。因为排序算法所需的辅助空间一般都不大，矛盾并不突出，而排序是经常执行的一种运算，往往属于系统的核心部分，因此，排序的时间开销是算法好坏的最重要的标志。排序的时间开销主要是指执行算法中关键字的比较次数和记录移动的次数，因此，在下面讨论各种内排序算法时，我们将给出各种算法的比较次数及移动次数。

8.2　插 入 排 序

插入排序(Insertion Sort)的基本思想：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的文件中的适当位置，直到全部记录插入完成为止。本节介绍直接插入排序和希尔排序。

8.2.1　直接插入排序

假设待排序的记录存放在数组 R 中，排序过程的某一中间时刻，R 被划分成两个子区间$[R[1], R[i-1]]$和$[R[i], R[n]]$，其中，前一个子区间是已排好序的有序区；后一个子区间则是当前未排序的部分，不妨称其为无序区。直接插入排序的基本操作是将当前无序区的

第 1 个记录 $R[i]$插入到有序区中适当位置，使得 $R[1]$到 $R[i]$变为新的有序区。

初始时，令 i=1，因为一个记录时自然是有序的，故 $R[1]$自成为一个有序区，无序区则是 $R[2]$到 $R[n]$，然后依次将 $R[2]$，$R[3]$，…插入到当前的有序区中，直至 i=n 时，将 $R[n]$插入到有序区为止。

但现在有一个问题，即如何将一个记录 $R[i]$(i=2，3，…，n)插入到当前的有序区，使得插入后仍保证该区间记录是按关键字有序的。显然，最简单的方法如下：首先，在当前有序区 $R[1]$到 $R[i-1]$中查找 $R[i]$的正确插入位置 $k(1\leqslant k\leqslant i-1)$；然后，将 $R[k]$到 $R[i-1]$中记录均后移一个位置，腾出 k 位置上的空间插入 $R[i]$。当然，若 $R[i]$的关键字大于 $R[1]$到 $R[i-1]$中所有记录的关键字，则 $R[i]$就插入原位置。但是，更为有效的方法是使查找比较操作和记录移动操作交替地进行，具体做法是将待插入记录 $R[i]$的关键字依次与有序区中的记录 $R[j]$(j=i−1，i−2，…，1)的关键字进行比较，若 $R[j]$的关键字大于 $R[i]$的关键字，则将 $R[j]$后移一个位置；若 $R[j]$的关键字小于或等于 $R[i]$的关键字，则查找过程结束，j+1 即为 $R[i]$的插入位置。因为关键字比 $R[i]$的关键字大的记录均已后移，所以 j+1 的位置已经腾空，只要将 $R[i]$直接插入此位置即可。下面给出其算法描述。

算法 8.1 直接插入排序。

```
void InsertSort(SeqList R)              /* 对数组 R 按递增序进行插入排序 */
{ /* R[0]是监视哨 */
int i,j;
   for(i=2;  i<=n;  i++)
   {
       R[0]=R[i];
       j=i-1;
       while (R[0].key<R[j].key)      /* 查找 R[i]的插入位置 */
          R[j+1]=R[j--];               /* 将关键字大于 R[i].key 的记录后移 */
       R[j+1]=R[0];                    /* 插入 R[i] */
    }
 }                                      /*插入排序*/
```


算法中引进附加记录 $R[0]$有两个作用：其一是进入查找循环之前，它保存了 $R[i]$的副本，使得不至于因记录的后移而丢失 $R[i]$中的内容；其二是在 while 循环“监视”下标变量 j 是否越界，一旦越界(即 $j<1$)，$R[0]$自动控制 while 循环的结束，从而避免了在 while 循环内的每一次都要检测 j 是否越界(即省略了循环条件“$j\geqslant 1$”)。因此，我们把 $R[0]$称为“监视哨”，这种技巧使得测试循环条件的时间大约减少一半，对于记录数较大的文件，节约的时间是相当客观的。

按照上述算法，我们用一个例子来说明直接插入排序的过程。设待排序的文件有八个记录，其关键字分别为 23、4、15、8、19、24、15'。为了区别两个相同的关键字 15，我们在后一个 15 右上加了一撇以示区别。其排序过程如图 8.1 所示，图中用括号表示当前的有序区。

直接插入排序算法由两重循环组成，对于有 n 个记录的文件外循环表示要进行 n−1 趟插入排序，内循环表明完成一趟排序所需进行的记录关键字间的比较和记录的后移。若初始文件按关键字递增有序(以下简称“正序”)，则在每一趟排序中仅需进行一次关键字的

比较，此时 $n-1$ 趟排序总的关键字比较次数取最小值 $C_{min}=n-1$；并且在每一趟排序中，无需后移记录。但是，在进入 while 循环之前，将 $R[i]$保存到监视哨 $R[0]$中需移动一次记录，在该循环结束之后将监视哨中 $R[i]$的副本插入到 $R[j+1]$也需要移动一次记录，此时排序过程总的记录移动此数也取最小值 $M_{min}=2(n-1)$。反之，若初始文件按关键字递增有序(以下简称“反序”)，则关键字的比较次数和记录移动次数均取最大值。在反序情况下，对于 for 循环的每一个 i 值，因为当前有序区 $R[1]$到 $R[i-1]$的关键字均大于待插入记录 $R[i]$的关键字，所以 while 循环中要进行 i 次比较才终止，并且有序区中所有的 $i-1$ 个记录均后移一个位置，再加上 while 循环前后的两次移动，则移动记录的次数为 $i-1+2$。由此可得排序过程关键字比较总次数的最大值 C_{max} 和记录移动总次数的最大值 M_{max}，即

$$C_{max}=\sum_{i=2}^{n} i=(n+2)(n-1)/2=O(n^2)$$

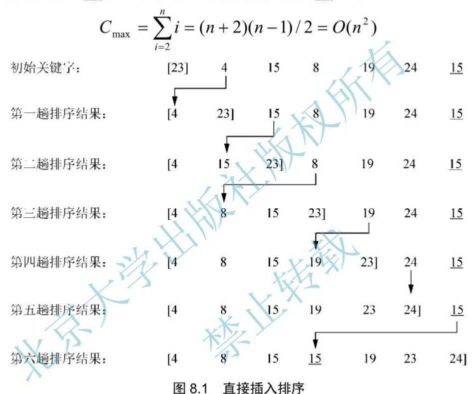


图 8.1　直接插入排序

$$M_{max}=\sum_{i=2}^{n}(i-1+2)=(n+1)(n+4)/2=O(n^2)$$

由上述分析可知，当文件的初始状态不同时，直接插入排序所耗费的时间是有很大差异的。最好情况是文件初始态为正序，此时算法的时间复杂度为 $O(n)$，最坏情况是文件初始态为反序，相应的时间复杂度为 $O(n^2)$。容易证明，算法的平均时间复杂度也是 $O(n^2)$。显然，算法所需的辅助空间是一个监视哨，故空间复杂度 $S(n)=O(1)$。

直接插入法排序是稳定的排序方法。

8.2.2　希尔排序

希尔排序(Shell’s Method)又称“缩小增量排序(Diminishing Increment)”，是由 D. L. Shell 在 1959 年提出来的。它的作用是先取定一个小于 n 的整数 d_1 作为第一个增量，把文件的全部记录分成 d_1 个组，所有距离为 d_1 倍数的记录放在同一个组中，在各组内进行直接插入排序；然后，取第二个增量 $d_2<d_1$ 重复上述分组和排序，直至所取的增量 $d_t=1(d_t<d_{t-1}<\cdots$

$<d_2<d_1$)，即所有记录放在同一组中进行直接插入排序为止。

先从一个具体例子来看希尔排序过程。假设待排序文件有 10 个记录，其关键字分别是 52，40，68，95，79，10，28，52*，58，06。增量序列取值依次为 5，3，1。

第一趟排序时，$d_1=5$，整个文件被分成五组：(R_1，R_6)，(R_2，R_7)，…，(R_5，R_{10})，各组中的第一个记录都自成一个有序区，我们依次将各组的记录 R_6，R_7，…，R_{10} 分别插入到各组的有序区中，使文件的各组均是有序的，其结果如图 8.2 所示的第七行。

第二趟排序时，$d_2=3$，整个文件分成(R_1，R_4，R_7，R_{10})，(R_2，R_5，R_8)，(R_3，R_6，R_9)，各组的第一个记录仍自成一个有序区，然后依次将各组的记录 R_4，R_5，R_6 分别插入到该组的当前有序区中，使得(R_1，R_4)，(R_2，R_5)，(R_3，R_6)均变为新的有序区，再依次将各组的记录 R_7，R_8，R_9 分别插入到该组的有序区中，又使得(R_1，R_4，R_7)，(R_2，R_5，R_8)，(R_3，R_6，R_9)均变为新的有序区，最后将 R_{10} 插入到有序区(R_1，R_4，R_7)中就得到第二趟排序结果。

第三趟排序时，$d_3=1$，即对整个文件做直接插入排序，其结果即为有序文件。

排序过程如图 8.2 所示。

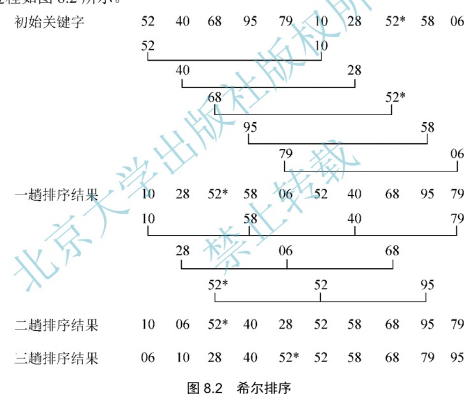


图 8.2 希尔排序

若不设置监视哨，根据上例的分析不难写出希尔排序算法，请读者自行完成。下面我们先分析如何设置监视哨，然后给出具体算法。设某一趟希尔排序的增量为 h，则整个文件被分成 h 组：(R_1，R_{h+1}，R_{2h+1}，…)，(R_2，R_{h+2}，R_{2h+2}，…)，…，(R_h，R_{2h}，R_{3h}，…)，因为各组中记录之间的距离均是 h，故第 1～h 租的哨兵位置依次为 $1-h$，$2-h$，…，0。如果像直接插入排序算法那样，将待插入记录 $R_i(h+1\leqslant i\leqslant n)$在查找插入位置之前保存到监视哨中，那么必须先计算 R_i 属于哪一组，才能决定使用哪个监视哨来保存 R_i。为了避免这种计算，我们可以将 R_i 保存到另一个辅助记录 x 中，而将所有监视哨 R_{1-h}，R_{2-h}，…，R_0 的关键字，设置为小写文件中的任何关键字。因为增量是变化的，所以各趟排序中所需的监视哨数目也不相同，但是可以按最大增量 d_1 来设置监视哨。具体算法描述如下。

算法 8.2 希尔排序。

```
void ShellSort(SeqList R,int d[])
{ /* R[0]到R[di-1]为di个监视哨, d[0]到d[t-1]为增量序列*/
    int i,j,k,h;
    rectype temp;
    int maxint=32767;                   /* 机器中最大整数 */
    for(i=0; i<d[0]; i++)
        R[i].key=-maxint;               /* 设置哨兵 */
    K=0;
    do {
    h=d[k];                             /* 取本趟增量 */
    for(i=h+di; i<n+d1; i++)            /* R[h+di]到R[n+di-1]插入当前有序区 */
    {
            temp=R[i];                  /* 保存待插入记录R[i] */
            j=i-n;
            while(temp.key<R[j].key)/* 查找正确的插入位置 */
            {
                R[j+h]=R[j];            /* 后移记录 */
                j=j-h;                  /* 得到前一记录 */
            }
                R[j+h]=temp;            /* 插入R[i] */
        }
        k++;
    }while(h!=1)                        /* 增量为1排序后终止算法 */
}
```


可能看出，当增量 h=1 时，ShellSort 算法与 InsertSort 基本一致。

人们对希尔排序的分析提出了许多困难的数学问题，特别是如何选择增量序列才能产生最好的排序效果，至今没有得到解决。读者可参考 D. E. Knuth 所著的《计算机程序设计技巧》第三卷，此书给出了希尔排序的平均比较次数和平均移动次数都在 $n^{1.3}$ 左右。

为什么希尔排序的时间性能优于直接排序插入排序呢？我们知道直接插入排序在文件初始态为正序时所需时间最少，实际上，当文件初始态基本有序时直接排序所需的比较和移动次数均较少。而当 n 值较少时，n 和 n^2 的差别也较小，即直接插入排序的最好时间复杂度 $O(n)$和最坏时间复杂度 $O(n^2)$差别不大。在希尔排序开始时增量较大，分组较多，每组的记录数目少，故各组内直接插入较快，当增量 d_i 逐渐缩小时，分组数直接减少，而各组的记录数目逐渐增多，但由于已经按 d_{i-1} 作为距离排过序，使文件较接近于有序状态，所以新的一趟排序过程也较快，因此，希尔排序在效率上较直接插入排序有较大的改进。

希尔排序是不稳定的。参见图 8.2，该例中两个相同关键字 49 在排序前后的相对次序发生了变化。

8.3 交换排序

换排序的基本思想：两两比较待排序记录的关键字，发现两个记录的次序相反时即进行交换，直到没有反序的记录为止。本节介绍两种交换排序：冒泡排序和快速排序。

8.3.1 冒泡排序

设想被排序的记录数组 $R[0]$到 $R[n-1]$垂直竖立，将每个记录 $R[i]$看作质量为 $R[i]$.key 的气泡。根据轻气泡不能在重气泡之下的原则，从上往下扫描数组 R，凡扫描到违反原则的轻气泡，就使其向上“漂浮”，如此反复进行，直到最后任何两个气泡都是轻者在上，重者在下为止。

初始时，$R[0]$到 $R[n-1]$为无序区，第一趟扫描从该区底部向上依次比较相邻两个气泡质量，若发现轻者在下，重者在上，则交换两者的位置。本趟扫描完毕时，“最轻”的气泡就漂浮到顶端，即关键字最小的记录被放在最高位置 $R[0]$上。第二趟扫描时，只需扫视 $R[1]$到 $R[n-1]$，扫描完毕时，“次轻”的气泡漂浮到 $R[1]$的位置上。一般地，第 i 趟扫描时，$R[0]$到 $R[i-1]$和 $R[i]$到 $R[n-1]$分别为当前的有序区和无序区，扫描仍是从无序区底部向上直至该区顶部，扫描完毕时，该区中最轻气泡漂浮到顶部位置 $R[i]$上，结果是 $R[0]$～$R[i]$变为新的有序区。图 8.3 是冒泡排序过程的示例，第 1 列为初始关键字，第 2 列起依次为各趟排序(即各趟扫描)结果，图中用方括号表示待排序的无序区。

初始关键字	第1趟结果	第2趟结果	第3趟结果	第4趟结果	第5趟结果	第6趟结果	第7趟结果
51	05	05	05	05	05	05	05
40	51	32	32	32	32	32	32
62	40	51	40	40	40	40	40
87	62	40	51	40*	40*	40*	40*
78	87	62	40*	51	51	51	51
05	78	87	62	62	62	62	62
32	32	78	87	78	78	78	78
40*	40*	40*	78	87	87	87	87

图 8.3 从下往上扫描的冒泡排序各趟结果

因为每一趟排序都使有序区增加一个气泡，在经过 $n-1$ 趟排序之后，有序区中就有 $n-1$ 个气泡，而无序区中气泡的质量总是大于等于有序区中气泡的质量，所以，整个起泡排序过程至多需要进行 $n-1$ 趟排序。但是，若在某一趟排序中未发现气泡位置的交换，则说明待排序的无序区中所有气泡均满足轻者在上，重者在下的原则，因此，起泡排序过程可在此趟排序后终止。在图 8.3 的示例中，在第五趟(图中第 6 列)排序过程中就没有冒泡交换位

置，此时整个文件已达到有序状态。为此，在下面给出的算法中，我们引入一个布尔量 noswap，在每趟排序之前，将它置为 TRUE。当一趟排序结束时，检查 noswap，若未曾交换过记录便终止算法。

算法 8.3 冒泡排序。

```
void BubbleSort(SeqList R)              /* 从下往上扫描的起泡排序 */
{
    int i,j,noswep;
    rectype temp;
    for(i=0; i<n-1; i++)                /* 做 n-1 趟排序*/
    {
        noswap=TRUE;                    /* 置未交换标志 */
        for(j=n-2; j>=i; j-- )          /* 从下往上扫描 */
        if(R[j+1].key<R[j].key)         /* 交换记录 */
        {
            temp=R[j+1];
            R[j+1]=R[j];
            R[j]=temp;
            noswap=FALSE;
        }
        if(noswap)   break;             /* 本趟排序中未发生交换,则终止算法 */
  }
}
```

容易看出，若文件的初始状态是正序，则一趟扫描就可完成排序，关键字的比较次数为 $n-1$，且没有记录移动。也就是说，起泡排序在最好情况下，时间复杂度是 $O(n)$。若初始文件是反序的，则需要进行 $n-1$ 趟排序，每趟排序要进行 $n-1$ 次关键字的比较($0 \leqslant i \leqslant n-2$)，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较移动次数均达到最大值，即

$$C_{\max} = \sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = O(n^2)$$

$$M_{\max} = \sum_{i=1}^{n-1}3(n-i) = 3n(n-1)/2 = O(n^2)$$

因此，起泡排序的最坏时间复杂度为 $O(n^2)$。它的平均时间复杂度也是 $O(n^2)$。显然，起泡排序是稳定的。

8.3.2 快速排序

快速排序(Quick Sort)又称划分交换排序。其基本思想是：在当前无序区 R[low]到 R[high]中任取一个记录作为比较的“基准”(不妨记为 temp)，用此基准将当前无序区划分为左右两个较小的无序子区：R[low]到 $R[i-1]$和 $R[i+1]$到 R[high]，且左边的无序子区中记录的关键字均小于或等于基准 temp 的关键字，右边的无序子区中记录的关键字均大于或等于基准 temp 的关键字，而基准 temp 则位于最终排序的位置上，即

R[low]到 $R[i-1]$中关键字≤temp.key≤$R[i+1]$到 R[high]的关键字(low≤i≤high)

当 R[low]到 $R[i-1]$和 $R[i+1]$到 R[high]均非空时，分别对它们进行上述的划分过程，直至所有无序子区中记录均已排好序为止。

要完成对当前无序区 R[low]到 R[high]的划分，具体做法是：设置两个指针 i 和 j，它们的初值分别为 i=low 和 j=high。不妨取基准为无序区的第 1 个记录 R[i](即 R[low])，并将它保存在变量 temp 中。令 j 自 high 起向左扫描，直到找到第 1 个关键字小于 temp.key 的记录 R[j]，将 R[j]移至 i 所指的位置上(这相当于交换了 R[j]和基准 R[i](即 temp)的位置，使关键字小于基准关键字的记录移到了基准的左边)；然后，令 i 自 i+1 起向右扫描，直至找到第 1 个关键字大于 temp.key 的记录 R[i]，将 R[i]移至 j 指的位置上(这相当于交换了 R[i]和基准 R[j](即 temp)的位置，使关键字大于基准关键字的记录移到了基准的右边)；接着，令 j 自 j-1 起向左扫描，如此交替改变扫描方向，从两端各自往中间靠拢，直至 i=j 时，i 便是基准 temp 的最终位置，将 temp 放在此位置上就完成了一次划分。

综合上面的叙述，下面给出排序的算法。

算法 8.4　快速排序

```
int PARTITION(SeqList R,low,high)
{/* 返回划分后被定位的基准记录的位置 */
    int i,j;
    RecType temp;
    i=low; j=high; temp=R[i];        /* 初始化 temp 为基准 */
    do{
        while((R[j].key>=temp.key) && (i<j))
                j--;    /*从右向左扫描,查找第一个关键字小于 temp.key 的记录*/
            if(i<j)  R[i++]=R[j];     /* 交换 R[i]和 R[j] */
            while((R[i].key<=temp.key) && (i<j))
                i++;    /*从左向右扫描,查找第 1 个关键字大于 temp.key 的记录*/
            if(i<j)   R[j--]=R[i];    /*交换 R[i]和 R[j]*/
        } while(i < j);
    R[i]=temp;                        /*基准 temp 已被最后定位*/
    return i;
}

void QuickSort(SeqList R,int low,int high)
{/*对 R[low]到 R[high]快速排序 */
    int i;
    if(low<high)                         /* 只有一个记录或无记录时无需排序*/
    {
        i=PARTITION(R,low,high);         /* 对 R[low]到 R[high]做划分 */
        QuickSort (R,low,i-1);           /*递归处理左区间*/
        QuickSort (R,i+1,high);          /*递归处理右区间*/
    }
}
```

注意：对整个文件 $R[0]$到 $R[n-1]$排序，只需调用 Quicksort(R，0，$n-1$)即可。

图 8.4 展示了一次划分的过程及整个快速排序的过程。图中方括号表示无序区，方框表示基准 temp 的关键字，它未参加真正的交换，只交在划分完成时才将它放到正确的位置上。

初始关键字	[(51)	40	62	87	78	05	32	40*	]
	↑i							↑j	
第1次交换后	[40*	40	62	87	78	05	32	(51)	]
		↑i						↑j	
i向右扫描	[40*	40	62	87	78	05	32	(51)	]
			↑i					↑j	
第2次交换后，i向左扫描	[40*	40	(51)	87	78	05	32	62	]
			↑i				↑j		
第3次交换后，i向左扫描	[40*	40	62	87	78	05	(51)	62	]
				↑i			↑j		
第4次交换后	[40*	40	32	(51)	78	05	87	62	]
				↑i		↑i			
第5次交换后	[40*	40	32	05	78	(51)	87	62	]
					↑i	↑j			
第6次交换后	[40*	40	32	05	(51)	78	87	62	]
					↑i ↑j				

(a) 一次划分过程

初始关键字	[51	40	62	87	78	05	32	40*]
1趟排序后	[40*	40	32	05]	51	[78	87	62]
2趟排序后	[40*	40	32	05]	51	[62]	78	[87]
3趟排序后	[05]	40	32]	40*	51	62	78	87]
4趟排序后	[05]	32	[40]	40*	51	62	78	87]
最终排序	05	32	40	40*	51	62	78	87

(b) 各趟排序后的状态

图 8.4 快速排序过程

对快速排序的性能做分析：最坏情况是每次划分选取的基准都是当前无序区中关键字最小(或最大)的记录，划分的结果是基准左边的无序子区为空(或右边的无序子区为空)，而划分所得的另一个非空的无序子区中记录数目，仅仅比划分前的无序区中记录个数减少 1。因此，快速排序必须做 $n-1$ 趟，每趟需进行 $n-i$ 次比较，故总的比较次数达到最大值：

$$C_{\max}=\sum_{i=1}^{n-1}(n-i)=n(n-1)/2=O(n^2)$$

显然，如果按上面给出划分算法，每次取当前无序区的第一个记录为基准，那么当文件的记录已按递增序(或递减序)排序时，每次划分所取的基准就是当前无序区中关键字最

小(或最大)的记录，则快速排序所需的比较次数反而最多。

在最好情况下，每次划分所取的基准都是当前无序区的“中值”记录，划分的结果是基准的左、右两个无序子区的长度大致相等。设 $C(n)$表示对长度为 n 的文件进行快速排序所需的比较次数，显然，它应该等于长度为 n 的无序区进行划分所需的比较次数 $n-1$ 加上递归地对划分所得的左、右两个无序子区(长度$\leqslant n/2$)进行快速排序所需的比较次数。假设文件长度 $n=2^k$，那么总的比较次数为：

$$
\begin{aligned}
C(n) &\leqslant n + 2C(n/2) \\
&\leqslant n + 2[n/2 + 2C(n/2^2)] = 2n + 4C(n/2^2) \\
&\leqslant 2n + 4[n/4 + 2C(n/2^3)] = 3n + 8C(n/2^3) \\
&\leqslant \cdots\cdots \\
&\leqslant kn + 2^k C(n/2^2) = n\log_2 n + nC(1) \\
&= O(n\log_2 n)
\end{aligned}
$$

注意：式中 $C(1)$为一个常数，$k=\log_2 n$。

因为快速排序的记录移动次数不大于比较的次数，所以，快速排序的最坏时间复杂度应为 $O(n_2)$，最好时间复杂度为 $O(n\log_2 n)$。为了改善最坏情况下的时间性能，可采用“三者取中”的规则，即在每一趟划分开始前，首先比较 R[low].key，R[high].key 和 $R[\lfloor(\text{low}+\text{high})/2\rfloor]$.key，令三者中取中值的记录和 R[low]交换。

可以证明：快速排序的平均时间复杂度也是 $O(n\log_2 n)$，它是目前基于比较的内部排序方法中速度最快的，快速排序亦因此而得名。

快速排序需要一个栈空间来实现递归。若每次划分均能将文件均匀分割为两部分，则栈的最大深度为$\lfloor \log_2 n \rfloor + 1$，所需栈空间为 $O(\log_2 n)$。最坏情况下，递归深度为 n，所需栈空间为 $O(n)$。

快速排序是不稳定的，请读者自行检验。

8.4 选 择 排 序

选择排序(Selection Sort)的基本方法：每一趟从待排序的记录中选出关键字最小的记录，顺序放在已排好序的子文件的最后，直到全部记录排序完毕为止。本节介绍两种选择排序：直接选择排序和堆排序。

8.4.1 直接选择排序

直接选择排序（StraightSelectionSort）的基本思想是：第一趟排序是在无序区 $R[0]$到 $R[n-1]$中选出最小的记录，将它与 $R[0]$交换；第二趟排序是无序区 $R[1]$到 $R[n-1]$中选关键字最小的记录，将它与 $R[1]$交换；而第 i 趟排序时 $R[0]$到 $R[i-2]$已是有序区，在当前的无序区 $R[i-1]$到 $R[n-1]$中选出关键字最小的记录 $R[k]$，将它与无序区中第 1 个记录 $R[i-1]$交换，使 $R[1]$到 $R[i-1]$变为新的有序区。因为每趟排序都使有序区中增加了一个记录，且有序区中的记录关键字不大于无序区中记录的关键字，所以进行 $n-1$ 趟排序后，整个文件就是递增有序的，直接选择排序的过程如图 8.5 所示，图中方括号表示无序区。

初始关键字	[52	40	68	95	79	10	28	52*	58	06]
第1趟排序后	06	[40	68	95	79	10	28	52*	58	52]
第2趟排序后	06	10	[68	95	79	40	28	52*	58	52]
第3趟排序后	06	10	28	[95	79	40	68	52*	58	52]
第4趟排序后	06	10	28	40	[79	95	68	52*	58	52]
第5趟排序后	06	10	28	40	52*	[95	68	79	58	52]
第6趟排序后	06	10	28	40	52*	52	[68	79	58	95]
第7趟排序后	06	10	28	40	52*	52	58	[79	68	95]
第8趟排序后	06	10	28	40	52*	52	58	68	[79	95]
最后排序结果	06	10	28	40	52*	52	58	68	79	95

图 8.5　直接选择排序

算法 8.5　直接选择排序。

```
void  SelectSort(SeqList R)       /*对R[0]到R[n-1]进行直接选择排序*/
{
    int i,j,k;
    RecType temp;
    for(i=0;i<n-1;i++)            /*做n-1趟选择排序*/
    {
    k=i;
        for(j=i+1; j<n;j++)       /*在当前无序区中选择关键字最小的记录R[k]*/
        if(R[j].key<R[k].key)  k=j;
            if(k!=i)   /*交换R[i]和R[k]*/
        {
            temp=R[i];
            R[i]=R[k];
            R[k]=temp;
        }
    }
}
```

显然，无论文件初始状态如何，在第 i 趟排序中选择最小关键字的记录，需做 $n-1$ 次比较，因此，总的比较次数为 $\sum_{i=1}^{n-1}(n-i)=n(n-1)/2=O(n^2)$。至于记录移动次数，当初始文件为正序时，移动次数为 0；文件初态为反序时，每趟排序均要执行交换操作，所以，总的移动次数取最大值 $3(n-1)$。直接选择排序的平均时间复杂度为 $O(n^2)$。

直接选择排序是不稳定的，请读者自行检验，如反例有 2，2'，1。

8.4.2　堆排序

在上一小节介绍的直接选择排序中，为了从 $R[1]$到 $R[n]$中选出关键字最小的记录，必须进行 n−1 次比较，然后在 $R[2]$到 $R[n]$选出关键字最小的记录，有需要做 n−2 次比较，事实上，后面这 n−2 次比较中，有许多比较可能在前面的 n−1 次比较中已经做过，但由于前一趟排序时未保留这些比较的结果，所以，后一趟排序时又重复执行了这些比较操作。本小节介绍的堆排序(Heap Sort)可以克服这一缺点。

堆排序是一种树形选择排序，它的特点是：在排序过程中，将 $R[1]$～$R[n]$看作一棵完全二叉树顺序存储结构，利用完全二叉树中双亲节点和孩子节点之间的内在关系(参见 6.2.3 小节)来选择关键字最小的记录。

首先，引出堆的定义，n 个关键字序列 K_1，K_2，…，K_n 称为堆，而且仅当该序列满足特性：

$$K_i \leqslant K_{2i} \quad 和 \quad K_i \leqslant K_{2i+1} \qquad (1 \leqslant i \leqslant \lfloor n/2 \rfloor)$$

从堆的定义可以看出，堆实质上是满足如下性质的完全二叉树：树中任一非叶子节点的关键字均小于或等于它的孩子节点的关键字。例如，关键字序列 12，20，45，26，35，85 就是一个堆，它所对的完全二叉树如图 8.6 所示。显然，这种堆中根节点(称为堆顶)的关键字最小，我们把它称为小根堆。反之，若将上面堆定义中的不等号反向，也就是说，若完全二叉树中任一非叶子节点的关键字均大于等于其孩子节点的关键字，则称之为大根堆，如图 8.7 所示，大根堆的堆顶关键字最大。显然，在堆中任一棵树亦是堆。

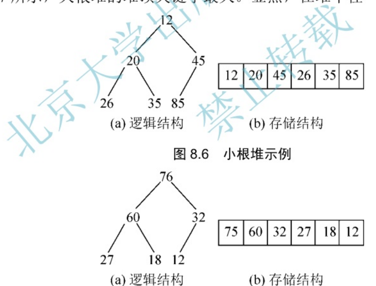


图 8.6　小根堆示例

图 8.7　大根堆示例

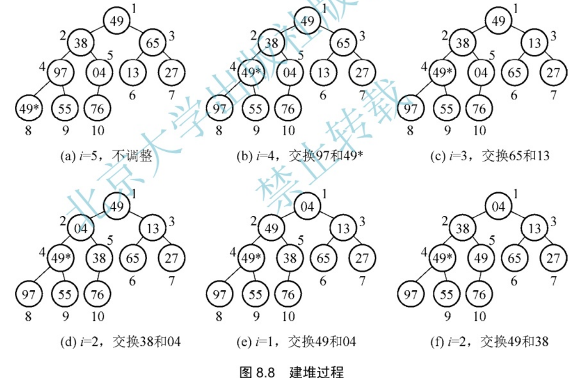
堆排序正是利用小根堆(或大根堆)来选取当前无序区中关键字最小(或最大)的记录实现排序的。我们不妨利用大根堆来排序。每一趟排序的基本操作如下：将当前无序区调整为一个大根堆时，选取关键字最大的堆顶记录，将它和无序区中最后一个记录交换。这样，正好和直接选择排序相反，有序区是在原记录区的尾部形成并逐渐向前扩大到整个记录区的。

堆排序的第一趟排序首先需“建堆”，即把整个记录数组 $R[1]$到 $R[n]$调整为一个大根堆，所以，必须把完全二叉树中以每一个节点为根的子树都调整为堆。显然只有一个节点的树是堆，而在完全二叉树中，所有序列 $i > \lfloor n/2 \rfloor$ 的节点都是叶子，因此，以这些节点为根的

子树均已是堆。这样，我们只需依次将以序列号为⌊$n/2$⌋，⌊$n/2$⌋-1，…，1，的节点作为根的子树都调整为堆即可。按该次序调整每个节点时，其左、右子树均已是堆(不妨将空树看作堆)。

现在的问题是，若已知节点 $R[i]$的左右子树均已是堆，如何将以 $R[i]$为根的完全二叉树也调整为堆？解决这一问题可采用“筛选法”。筛选法的基本思想如下：因为 $R[i]$的左、右子树已是堆，这两棵子树的根分别是各自子树中的最大关键字，所以，我们必须在 $R[i]$和它的左、右孩子中选取关键字最大的节点放到 $R[i]$的位置上。若 $R[i]$的关键字已是三者中的最大者，则无需做任何调整，以 $R[i]$为根的子树已构成堆；否则，必须将 $R[i]$和具有最大关键字的左孩子 $R[2i]$或 $R[2i+1]$进行交换。不妨设 $R[2i]$的关键字最大，将 $R[i]$和 $R[2i]$交换位置，交换之后有可能导致以 $R[2i+1]$为根的子树不再是堆，但由于 $R[2i]$的左、右子树仍然是堆，于是可重复上述过程，将以 $R[2i]$为根的子树调整为堆，……，如此重复，逐层递推下去，最多可能一直调整到树叶，最终把最小的关键字筛选下去，将最大关键字一层层地选择上来。

图 8.8 阐明了对于关键字序列 49，38，65，97，04，13，27，49*，55，76 在建堆过程中完全二叉树及其存储结构的变化情况，其中 n=10，故从第 5 个节点起进行调整。



(a) i=5，不调整 (b) i=4，交换97和49* (c) i=3，交换65和13

(d) i=2，交换38和04 (e) i=1，交换49和04 (f) i=2，交换49和38

图 8.8 建堆过程

算法 8.6 筛选算法。

```
void Sift(SeqList R, int i, int m)   /* 在数组R[i]到R[m]中,调整R[i] */
{  /* 以R[i]为根的完全二叉树构成堆 */
    int j;
    RecType temp;
    temp=R[i];
    j=2*i;
    while(j<=m)        /* j≤m,R[2*i]是R[i]的左孩子 */
```

```
        {
            if((j<m) && (R[j].key<R[j+1].key)  j++;  /* j 指向 R[i]的右孩子 */
            if(temp.key<R[j].key)     /* 孩子节点关键字较大 */
            {
                R[i]=R[j];            /* 将 R[j]换到双亲位置上 */
                i=j;  j=2*i           /* 修改当前被调整节点 */
            }
            else break;               /* 调整完毕,退出循环 */
        }
        R[i]=temp;                    /* 最初被调整节点放入正确位置 */
    }
```

在完全二叉树中，若一个节点没有左孩子，则该节点必是叶子，因此，当筛选算法中循环条件 $j \leq m$ 不成立时，则表示当前调整节点 $R[i]$是叶子，故筛选过程可以结束。在筛选过程中，若当前被调整节点 $R[i]$和它的左、右孩子节点相比，某一孩子 $R[j]$的关键字最大，则需要交换 $R[i]$和 $R[j]$的位置，将 $R[i]$筛选至下一层，但由于 $R[i]$还可能会被逐层筛下去，为了减少记录移动次数，故算法在筛选开始前将最初被调整的节点 $R[i]$保存在 temp 中，当发生交换时，仅需将 $R[j]$放入其双亲节点 $R[i]$的位置上，而 $R[i]$未直接放入 $R[j]$的位置上，只有当整个筛选过程结束时，才将其保存在 temp 中的记录放到最重要的位置上。

有了上述筛选算法，则将最初的无序区 $R[1]$到 $R[n]$建成一个大根堆，可用下面语句实现：

```
        for(i=n/2; i>=1; i--)
          Sift(R,i,n);
```


由于建堆的结果是把 $R[1]$到 $R[n]$中关键字最大的记录筛选到堆顶 $R[1]$的位置上，排序后这个关键字最大的记录应该是记录区 $R[1]$到 $R[n]$的最后一个记录，因此，将 $R[1]$和 $R[n]$交换后便得到了第一趟排序的结果。

第二趟排序的操作首先是将当前无序区 $R[1]$到 $R[n-1]$调整为堆。因为第一趟排序后，$R[1]$到 $R[n]$中只有 $R[1]$的值发生了变化，它的左、右孩子仍然是堆，所以，我们可以调用 Sift(R，1，$n-1$)将 $R[1]$到 $R[n-1]$调整为大根堆，即选出 $R[1]$到 $R[n-1]$中最大关键字放入堆顶。然后，将堆顶记录 $R[1]$和当前无序区的最后一个记录 $R[n-1]$交换，其结果是 $R[1]$到 $R[n-2]$变为新的无序区，$R[n-1]$到 $R[n]$为有序区，且有序区中记录的关键字均大于等于无序区中记录的关键字。如此重复 $n-1$ 趟排序之后，就使有序区扩充到整个记录区 $R[1]$到 $R[n]$。图 8.9 是堆排序的全过程示例，其中方括号里面的记录是当前的无序区。

算法 8.7　堆排序。

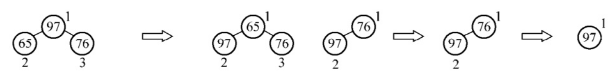
```
    void HeapSort (SeqList R)
    { /* 对 R[1]到 R[n]进行堆排序 */
        int i;
        RecType temp;
        for(i=n/2;i>=1;i--)        /*建初始堆*/
            Sift(R,i,n);
```

```
        for(i=n;i>=1;i--)
        {
            temp=R[1];              /*当前堆顶记录和最后一个记录交换*/
            R[1]=R[i];
            R[i]=temp;
            Sift(R,1,i-1);    /*R[1]到R[i-1]重建成堆 */
        }
    }
```

(a) 输出04，将76移至堆顶，重新整堆　　(b) 输出13，将55移至堆顶，重新整堆

(c) 输出27，将97移至堆顶，重新整堆　　(d) 输出38，将76移至堆顶，重新整堆

(e) 输出49*，将65移至堆顶，重新整堆　　(f) 输出49，将76移至堆顶，重新整堆



(g) 输出55，将97移至堆顶，重新整堆　　(h) 输出65，76，97

图 8.9　堆排序过程

堆排序的时间主要由建立初始堆和不断重建这两部分的时间开销构成。建立初始堆共调用了 Sift 过程$\lfloor (n/2) \rfloor$次，每次均是将 $R[i]$为根 ($\lfloor n/2 \rfloor \geqslant i \geqslant 1$) 的子树调整为堆。显然，具有 n 个节点的完全二叉树深度是 $h = \lfloor \log_2 n \rfloor + 1$，故节点 $R[i]$ ($\lfloor n/2 \rfloor \geqslant i \geqslant 1$) 的层数只可能是 $h-1, h-2, \cdots, 1$。由于第 l 层上的节点个数至多为 2^{l-1}，故以它们为根的子树深度为 $h-l-1$。而 Sift 算法对深度为 k 完全二叉树所进行的关键字比较次数，至多为 $2(k-1)$次，因此，建初始堆调用 Sift 算法所进行的关键字比较的总次数，不超过 $C_1(n)$，且它满足下式：

$$
\begin{aligned}
C_1(n) &= \sum_{i=h-1}^{1} 2^{i-1} \times 2(h-1) \\
&= \sum_{i=h-1}^{1} 2^{i} \times (h-1) \\
&= 2^{h-1} + 2^{h-2} \times 2 + 2^{h-3} \times 3 + \cdots + 2 \times (h-1) \\
&= 2^{h}(1/2 + 2/2^2 + 3/2^3 \cdots + (h-1)/2^{h-1}) \\
&\leqslant 2^{h} \times 2 \leqslant 2 \times 2^{(\log_2 n)+1} = 4n \\
&= O(n)
\end{aligned}
$$

第 j 次重建堆时，堆中有 $n-j$ 个节点，完全二叉树深度为$[\log_2(n-j)]+1$，调用 Sift 重建堆所需的比较次数至多为$2\times[\log_2(n-j)]$。因此，$n-1$ 趟排序过程中重建堆的比较总次数不超过 $C_2(n)$，且

$$
\begin{aligned}
C_2(n) &= 2 \times ([\log_2(n-1)] + [\log_2(n-2)] + \cdots + [\log_2 2]) \\
&< 2n[\log_2 n] \\
&= O(n\log_2 n)
\end{aligned}
$$

在 Sift 算法中，记录移动次数不会超过比较次数，因此，堆排序的时间复杂度是$O(n+n\log_2 n)=O(n\log_2 n)$。

由于建初始堆所需的比较次数较多，所以，堆排序不适宜于记录数较少的文件。堆排序是我们介绍的第一个最坏时间复杂度$O(n\log_2 n)$的排序算法，辅助存储空间仅为用于交换的记录空间。

堆排序是不稳定的，请读者自己举出反例。

8.5 编程实现奥运会奥运奖牌排名系统

(1) 数据结构定义如下。

```
#define MaxSize 100
typedef struct
{  char number[20];                /*英文缩写*/
    char name[20];                 /*国家名称*/
     int gold;                     /*金牌数*/
     int silver;                   /*银牌数*/
     int copper;                   /*铜牌数*/
     int total;                    /*总牌数*/
     int place;                    /* 名次*/
} DataType;                        /*国家信息类型*/

typedef struct
{
     DataType data[MaxSize];       /* 存储各国家信息的数组*/
     int length;                   /*国家数*/
}CountryList ;                     /*顺序表,存储国家列表*/
```

(2) 主程序代码如下

```
void select2(CountryList *L) ;
void select1(CountryList *L) ;
int a[MaxSize];                          //存储金银铜牌的加权和
void PlaceSetting(CountryList *L)        //设置排名
{
    int i,rankid=1;
    for(i=1,a[0]=0;i<=L->length;i++,rankid++)    //金银铜牌相同时排名相同
        if(a[i]==a[i-1])
            L->data[i].place=--rankid;
        else
            L->data[i].place=rankid;
}

void WeightSumComp(CountryList *L)   //将金银铜牌进行加权和运算
{
    int i;
    for(i=1;i<=L->length;i++)
    {
        a[i]=L->data[i].gold*10000+L->data[i].silver*100+L->data[i].
copper;
    }
}

void FCreat(CountryList *L)              //读取信息
{   FILE *fp;
    int i=1;
    fp=fopen("shuju.txt","r");           //以只读的方式打开 shuju.txt
    if(!fp)
        printf("\n 数据文件无法打开!(Can not open file!)");
    else
        {
            L->length=0;
            while(!feof(fp))
            {
                fscanf(fp,"%d%s%s%d%d%d%d\n",&(L->data[i].place),
L->data[i].name,L->data[i].number,&(L->data[i].gold),&(L->data[i].silver),&(
L->data[i].copper),&(L->data[i].total));
                L->length++;
                i++;
            }
        }
}
```

```
void sortJYT(CountryList *L)          //按金银铜牌数排序
{
    int i,j;
    for(i=1;i<L->length;i++)
        for(j=i+1;j<=L->length;j++)
            if(a[i]<a[j])
            {
                a[0]=a[i];a[i]=a[j];a[j]=a[0];
                L->data[0]=L->data[i];
               L->data[i]=L->data[j];
               L->data[j]=L->data[0];
            }
            else

    if(a[i]==a[j]&&strcmp(L->data[i].number,L->data[j].number)>0)
                {
                     L->data[0]=L->data[i];
                     L->data[i]=L->data[j];
                     L->data[j]=L->data[0];
                }
    PlaceSetting(L);
}

void sortALL(CountryList *L)          //按总牌数排序
{
    int i,j;
    for( i=1;i<L->length;i++)
        for(j=i+1;j<=L->length;j++)
            if(L->data[i].total < L->data[j].total)
            {
                a[0]=a[i];a[i]=a[j];a[j]=a[0];
                L->data[0]=L->data[i];
                L->data[i]=L->data[j];
                L->data[j]=L->data[0];
            }
            else if((L->data[i].total==L->data[j].total) && a[i]<a[j])
            {
                a[0]=a[i];a[i]=a[j];a[j]=a[0];
                L->data[0]=L->data[i];
                L->data[i]=L->data[j];
                L->data[j]=L->data[0];
            }
```

```
        PlaceSetting(L);
    }

    void output(CountryList *L)          //输出排名
    {
        char a='0'; int i;
        printf("\n 排名国家和地区英文缩写金银铜总数\n");
        for(i=1;i<=L->length;i++)
        {
            printf("%2d%17s%13s",(L->data[i].place),L->data[i].name,L->>
data[i].number);
            if(L->data[i].gold)
                printf("%11d",L->data[i].gold);
            else
                printf("%11c",a);
            if(L->data[i].silver)
                printf("%6d",L->data[i].silver);
            else
                printf("%6c",a);
            if(L->data[i].copper)
                printf("%6d",L->data[i].copper);
            else
                printf("%6c",a);
             printf("%7d\n",(L->data[i].total));
        }
    }
    /*下面是七种排序方法*/
    void InsertSort(CountryList *L)          //直接插入排序
    {
        int i,j;
        for(i=2;i<=L->length;i++)
        if(L->data[i].total>L->data[i-1].total)
        {    L->data[0]=L->data[i];
             j=i-1;
             while(L->data[0].total>L->data[j].total)
             {
                 L->data[j+1]=L->data[j];
                 j--;
             }
             L->data[j+1]=L->data[0];
        }
        PlaceSetting(L);
    }
```

```
void ShellSort(CountryList *L)          //希尔排序
{
    int i,j,d=L->length;
    do{
        d=d/3+1;                        //d 为当前增量,求下一个增量的方法不唯一
        for(i=d+1;i<=L->length;i++)
        if(L->data[i].total>L->data[i-d].total)
        {
            L->data[0]=L->data[i];   //L->data[0]为暂存单元,不为哨兵
            j=i-d;
            while(j>0&&L->data[0].total>L->data[j].total)
            {
                L->data[j+d]=L->data[j];
                j=j-d;
            }
            L->data[j+d]=L->data[0];
        }
    }while(d!=1);                      //增量为 1 排序后终止
    PlaceSetting(L);
}

void BubbleSort(CountryList *L)        //冒泡排序
{
    int i,j;
    for(i=1; i<L->length; i++)
    {
        int noswap=1;                   //置未交换标志为真
        for(j=L->length-1;j>=i;j--)
        {
        if(L->data[j+1].total>L->data[j].total)
        {
                L->data[0]=L->data[j+1];
                L->data[j+1]=L->data[j];
                L->data[j]=L->data[0];
                noswap=0;               //若已交换,则置未交换标志为否
            }
    }
        if(noswap)
          break;
    }
    PlaceSetting(L);
}
```

```
int Partition(CountryList *L,int m,int n)       //快速排序
{
    CountryMedals temp;
    int i,j;
    i=m; j=n; temp=L->data[i];
            do{
                while((L->data[j].total<=temp.total) && (i<j))
                 j--;
                if(i<j) L->data[i++]=L->data[j];
                while((L->data[i].total>=temp.total) && (i<j))
                 i++;
                if(i<j)   L->data[j--]=L->data[i];
            } while(i != j);
            L->data[i]=temp;
            return i;
}

void QuickSort(CountryList *L,int low,int high)
{
    int pos,i;
    if(low<high)
    {
        pos=Partition(L,low,high);
        QuickSort(L,low,pos-1);
        QuickSort(L,pos+1,high);
    }
    PlaceSetting(L);
}

void SelectSort(CountryList *L)          //直接选择排序
{
    int i,j,k;
    for(i=1;i<L->length;i++)
    {
        k=i;
        for(j=i+1;j<=L->length;j++)      //在无序区找到最大记录的位置
            if(L->data[j].total>L->data[k].total)
                k=j;
        if(k!=i)
        {
            L->data[0]=L->data[i];
            L->data[i]=L->data[k];
```

```
                L->data[k]=L->data[0];
            }
        }
        PlaceSetting(L);
}

void Sift(CountryList *L,int i,int m)             //堆排序
{
        int j;
        L->data[0]=L->data[i];
        j=2*i;
        while(j<=m)
        {
            if(j<m && L->data[j].total>L->data[j+1].total)
                j++;
            if(L->data[0].total>L->data[j].total)
            {
                L->data[i]=L->data[j];
                i=j;
                j=2*i;
            }
            else break;
        }
        L->data[i]=L->data[0];
}

void HeapSort(CountryList *L)
{
        int i;
        for(i=L->length/2;i>=1;i--)          //建立初始堆
            Sift(L,i,L->length);
        for(i=L->length;i>1;i--)
        {
            L->data[0]=L->data[1];
            L->data[1]=L->data[i];
            L->data[i]=L->data[0];
            Sift(L,1,i-1);
        }
        PlaceSetting(L);
}

void select1(CountryList *L)                //主菜单
{
```

```
    system("cls");
        puts("****第 29 届奥运会奥运奖牌的排名情况*****");
        puts("*      1-按金牌总数排名              *");
        puts("*      2-按奖牌总数排名              *");
        puts("*      3-六种排序时间比较(按奖牌总数)*");
        puts("*      4-显示排行榜                  *");
        puts("*      0-退出系统                    *");
        puts("****************************************");
        printf("请选择输入: ");
        switch(getche())
        {

            case '1':
                sortJYT(L);
                output(L);
                printf("排序完成~~\n");
                printf("按任意键返回~~\n");
                getch();
                break;
             case '2':
                sortALL(L);
                output(L);
                printf("排序完成~~\n");
                printf("按任意键返回~~\n");
                getch();
                break;
             case '3':
                select2(L);
                break;
             case '4':
                output(L);
                printf("输出完毕~~~\n");
                printf("按任意键返回~~\n");
                getch();
                break;
             case '0': exit(0);
        }
}

void select2(CountryList *L)         //子菜单
{
        double time1,time2;
        puts("\n==========按奖牌总数七种排序===========");
```

```
puts("++     1-直接插入排序(稳定)          ++");
puts("++     2-希尔排序(不稳定)            ++");
puts("++     3-冒泡排序(稳定)              ++");
puts("++     4-快速排序(不稳定)            ++");
puts("++     5-直接选择排序(不稳定)        ++");
puts("++     6-堆排序(不稳定)              ++");
puts("++     0-返回上层                    ++");
puts("========================================");
switch(getche())
{
    case '1':
         time1=(double)clock()/CLOCKS_PER_SEC;
         //clock()/CLOCKS_PER_SEC 为程序开始运行时计时的函数,单位为 s
         InsertSort(L);
        time2=(double)clock()/CLOCKS_PER_SEC;
         break;
     case '2':
         time1=(double)clock()/CLOCKS_PER_SEC;
         //估计数据量少,排序时间基本为零,但加上输出函数时时间增加
         ShellSort(L);
         time2=(double)clock()/CLOCKS_PER_SEC;
         break;
     case '3':
         time1=(double)clock()/CLOCKS_PER_SEC;
         BubbleSort(L);
         time2=(double)clock()/CLOCKS_PER_SEC;
         break;
     case '4':
         time1=(double)clock()/CLOCKS_PER_SEC;
     QuickSort(L,1,L->length);
         time2=(double)clock()/CLOCKS_PER_SEC;
         break;
     case '5':
         time1=(double)clock()/CLOCKS_PER_SEC;
         SelectSort(L);
         time2=(double)clock()/CLOCKS_PER_SEC;
         break;
     case '6':
         time1=(double)clock()/CLOCKS_PER_SEC;
         HeapSort(L);
         time2=(double)clock()/CLOCKS_PER_SEC;
         break;
     case '0': select1(L);
```

```
        }
        output(L);
        printf("排序完!所用时间 time=%.3lf\n",(time2-time1));
        printf("按任意键返回~~\n");
        getch();
    }

    void main()
    {
        CountryList countrylist;
        FCreat(&countrylist);
        WeightSumComp(&countrylist);
        while(1)
            select1(&countrylist);
    }
```



独立实践

请读者对上述程序中排序算法运行时间的计算功能进行改进。

小　　结

本章主要讨论各种常见的内部排序方法的原理和设计实现，并对这些排序算法的稳定性和复杂性进行了较为详尽的分析。可以看出，每一种排序方法都有其优缺点，有其本身适用的场合，应该按照具体情况进行合理的选用。

在选择排序方法时，有下列几种选择：

(1) 若待排序的记录个数 n 值较小，可以选用直接插入排序法，但是若记录所含数据项较多，所占存储量较大时，应选用直接选择排序法。反之，若待排序的记录个数 n 值较大时，应选用快速排序法。

(2) 快速排序在 n 值较小时的性能不及直接插入排序，因此在实际应用中，可将它和直接插入排序混合使用。如在快速排序划分子区间的长度小于某值时，转而调用直接插入排序。

(3) 若待排序序列总是基本有序，用冒泡排序、简单选择排序或直接插入排序更适合，不适合用快速排序。

习　　题

一、填空题

1．大多数排序算法都有两个基本的操作：________和________。

2．在对一组记录(54，38，96，23，15，72，60，45，83)进行直接插入排序时，当把第七个记录 60 插入到有序表时，为寻找插入位置至少需比较________次。

3．在插入和选择排序中，若初始数据基本正序，则选用________；若初始数据基本反序，则选用________。

4．在堆排序和快速排序中，若初始记录接近正序或反序，则选用________；若初始记录基本无序，则最好选用________。

5．对于 n 个记录的集合进行冒泡排序，在最坏的情况下所需要的时间是________。若对其进行快速排序，在最坏的情况下所需要的时间是________。

6．设要将序列(Q，H，C，Y，P，A，M，S，R，D，F，X)中的关键码按字母序的升序重新排列，则冒泡排序一趟扫描的结果是________；

初始步长为 4 的希尔排序一趟的结果是________；

快速排序一趟扫描的结果是________；

堆排序初始建堆的结果是________。

二、判断题

1．用希尔方法排序时，若关键字的初始排序杂乱无序，则排序效率偏低。 ( )

2．堆排序所需要附加空间数与待排序的记录个数无关。 ( )

3．对 n 个记录的集合进行快速排序，所需要的平均时间是 $O(n\log_2 n)$。 ( )

4．快速排序的速度在所有排序方法中为最快，而且所需附加空间也最少。 ( )

5．对一个堆，按二叉树层次进行遍历可以得到一个有序序列。 ( )

6．对于 n 个记录的集合进行冒泡排序，所需的平均时间是 $O(n)$。 ( )

三、选择题

1．将五个不同的数据进行排序，至多需要比较( )次。

A．8　　B．9　　C．10　　D．25

2．排序方法中，从未排序序列中依次取出元素与已排序序列(初始时为空)中的元素进行比较，将其放入已排序序列的正确位置上的方法称为( )。

A．希尔排序　　B．冒泡排序　　C．插入排序　　D．选择排序

3．从未排序序列中挑选元素，并将其依次插入已排序序列(初始时为空)的一端的方法称为( )。

A．希尔排序　　B．堆排序　　C．插入排序　　D．选择排序

4．对 n 个不同的排序码进行冒泡排序，在下列情况下比较的次数最多的是( )。

A．从小到大排列好的　　B．从大到小排列好的

C．元素无序　　D．元素基本有序

5．对 n 个不同的排序码进行冒泡排序，在元素无序的情况下比较的次数为( )。

A．$n+1$　　B．n　　C．$n-1$　　D．$n(n-1)/2$

6．快速排序在下列情况下最易发挥其长处的是( )。

A．被排序的数据中含有多个相同排序码

B．被排序的数据已基本有序

C．被排序的数据完全无序

D．被排序的数据中的最大值和最小值相差悬殊

7. 对有 n 个记录的表做快速排序，在最坏情况下，算法的时间复杂度是(　　)。

A．$O(n)$　　B．$O(n^2)$　　C．$O(n\log_2 n)$　　D．$O(n^3)$

8. 若一组记录的排序码为(46，79，56，38，40，84)，则利用快速排序的方法，以第一个记录为基准得到的一次划分结果为(　　)。

A．38，40，46，56，79，84　　B．40，38，46 ，79，56，84

C．40，38，46，56，79，84　　D．40，38，46，84，56，79

9. 下列关键字序列中，(　　)是堆。

A．16，72，31，23，94，53　　B．94，23，31，72，16，53

C．16，53，23，94，31，72　　D．16，23，53，31，94，72

10. 堆是一种(　　)排序。

A．插入　　B．选择　　C．交换　　D．归并

11. 堆的形状是一棵(　　)。

A．二叉排序树　　B．满二叉树　　C．完全二叉树　　D．平衡二叉树

12. 若一组记录的排序码为(46，79，56，38，40，84)，则利用堆排序的方法建立的初始堆为(　　)。

A．79，46，56，38，40，84　　B．84，79，56，38，40，46

C．84，79，56，46，40，38　　D．84，56，79，40，46，38

13. 下述几种排序方法中，要求内存最大的是(　　)。

A．插入排序　　B．快速排序　　C．堆排序　　D．选择排序

四、应用题

1. 已知关键字序列{98，82，105，71，36，77，24，82'，12，55}，分别写出直接插入排序、希尔排序(增量为5，3，1)、冒泡排序、快速排序、直接选择排序、堆排序的各趟运行结果。

2. 以单链表为存储结构实现直接插入排序的算法。

3. 修改冒泡排序算法，以交替的正、反两个方向进行扫描。即第一趟把排序码最大的记录放到最末尾，第二趟把排序码最小的记录放到最前面。如此反复进行直到排序完成。

4. 以单链表为存储结构，写一个直接选择排序算法。

5. 一个线性表中的元素为正整数或负整数，设计一个算法，将正整数和负整数分开，使线性表的前部为负整数，后部为正整数，不要求都它们排序，但要求尽量减少交换次数。



查　　找

问题描述

电话号码查询系统

人们在日常生活中经常需要查询某个人或某个单位的电话号码，一个简单、易用、快捷的电话号码查询系统能给人们生活带来极大地方便。要求实现一个简单的个人电话号码查找系统，根据用户输入的信息(如姓名等)进行快速查询，要求能够完成号码的录入和查找功能。每个记录有电话号码、用户名和地址 3 个信息。

9.1　概　　述

在利用计算机进行数据处理时，特别是在非数值处理中，查找(又称检索)是最常用的一种操作，几乎在任何一个计算机系统软件和应用软件中都会涉及，当问题所涉及的数据量相当大时，查找方法的效率就显得格外重要，在一些实时应答系统中尤其重要，因此有必要掌握一些常用的查找方法，并通过对它们的效率分析来比较各种查找方法的优劣。

在计算机领域中，“查找”有明确而严格的定义，下面给出有关的概念。

查找表(Search Table)：由同一类型的数据元素(或记录)构成的集合，即查找表中的数据元素具有相同的类型。根据对查找表的操作不同分类，查找表分为静态查找表和动态查找表。

静态查找表(Static Search Table)：若只对查找表进行查询(查询某个“特定的”数据元素是否在查找表中)和检索(获取指定数据元素的各种信息)操作，则这类查找表称为静态查找表。

动态查找表(Dynamic Search Table)：若在查找过程中同时插入查找表中不存在的数据元素，或者从表中删除已经存在的某个数据元素，则这类查找表称为动态查找表。

关键字(Key)：数据元素中某个数据项的值，又称为键值，可以用来标识一个数据元素。

关键码：关键字所在的某个数据项(或称字段)。

主关键字：能够唯一标识数据元素(或记录)的关键字。

次关键字：不能够唯一标识数据元素(或记录)的关键字。例如，在电话号码本中，能够唯一标识一条记录的是电话号码，即主关键字；而用户名或地址均不能唯一标识一条记录，因此它们是次关键字。

查找(Searching)：在含有 n 条记录的表中找出关键字等于给定值 K 的数据元素(记录)。若找到，则查找成功，返回该记录的信息或该记录在表中的位置；否则查找失败，返回相

关的指示信息。

因为查找是对已存入计算机中的数据所进行的操作，所以采用何种查找方法，首先取决于使用哪种数据结构来表示“表”，即表中节点是何种方式组织的，为了提高查找速度，我们常常用某些特殊的数据结构来组织表。因此，在研究各种查找方法时，首先必须弄清这些方法所需要的数据结构，特别是存储结构。

和排序类似，查找也有内查找和外查找之分。若整个查找过程都在内存中进行，则称之为内查找；反之，若查找过程中需要访问外存，则称之为外查找。

由于查找运算的主要操作是关键字的比较，所以，通常将查找过程中对关键字需要执行的平均比较次数(也称平均查找长度)作为衡量一个查找算法效率优劣的标准。平均查找长度(Average Search Length，ASL)定义为：

$$\mathrm{ASL}=\sum_{i=1}^{n} p_i c_i \tag{9.1}$$

式中：n 是节点的个数；p_i 是查找第 i 个节点的概率，若不特别声明，均认为每个节点的查找概率相等，即 $p_1=p_2=\cdots=p_n=1/n$；c_i 是找到第 i 个节点所需要的比较次数。

9.2 线性表查找

在表的组织方式中，线性表是最简单的一种，本节将介绍三种在线性表上进行查找的方法，它们分别是顺序查找、二分查找和分块查找。

9.2.1 顺序查找

顺序查找是一种最简单的查找方法。它的基本思想如下：从表的一端开始，顺序扫描线性表，依次将扫描到的节点关键字和给定值 K 相比较，若当前扫描到的节点关键字与 K 相等，则查找成功；若扫描结束后，仍未找到关键字等于 K 的节点，则查找失败。

假设查找表中的关键字为{34，44，43，12，53，55，73，64，77}，如果待查关键字为 64，则从 34 开始向后比较，比较到 64 时查找成功，或从 77 开始向前比较，比较到 64 时查找成功。而若查找关键字为 88，则从 34 开始向后或从 77 开始向前比较，比较完所有元素后都没有找到相等的记录，查找失败。

顺序查找方法既适用于线性表的顺序存储结构，也适用于线性表的链式存储结构。使用单链表做存储结构时，扫描必须从第一个节点开始往后扫描。下面只介绍以数组做存储结构时的顺序查找，类型说明和具体算法如下。

```
typedef struct{
    KeyType key;          /*KeyType 由用户定义*/
    InfoType otherinfo;   /*此类型依赖于应用*/
    }NodeType;
typedef NodeType Sqlist[n+1];
```

算法 9.1 顺序查找算法。

```
int SqSearch(Sqlist R,KeyType K)
```

```
{/*在顺序表 R[1..n]中顺序查找关键字为 K 的记录*/
    /*成功时返回找到的记录位置,失败时返回 0*/
    int i;
    R[0].key=K;                      /*在 0 位置设置监视哨*/
    for(i=n;R[i].key!=K;i--);        /*从表后往前找*/
    return i;
    /*若 i 为 0,表示查找失败,否则 R[i]为要找的记录*/
}
```

算法中监视哨 $R[0]$的作用仍然是在 for 循环中省去判定防止下标越界的条件 $i\geqslant1$，从而节省比较的时间。若从后向前扫描完整个表后，都未找到关键字为 K 的节点，则循环必终止于 $R[0].key=K$，此时返回的函数值为 0，这意味着查找失败；若 for 循环终止时，$i\geqslant1$，则查找成功。显然，若找到的是 $R[n]$，则比较次数 $c_n=1$；若找到的是 $R[1]$，则比较次数 $c_1=n$；一般情况下，$c_i=n-i+1$，因此，在等概率假设下，顺序查找的平均查找长度为：

$$ASL_{sq}=\sum_{i=1}^{n}p_ic_i=\sum_{i=1}^{n}i/n=(n+1)/2 \tag{9.2}$$

也就是说，查找成功的平均比较次数约为表长度的一半。若 K 值不在表中，则必须进行 $n+1$ 次比较之后才能确定查找失败。

顺序查找算法中的基本工作就是关键字的比较，因此，查找长度的数量级就是查找算法的时间复杂度，记为 $O(n)$。

有时，表中各节点的查找概率并不相等，例如，在由全体学生的病历档案组成的线性表中，体弱多病学生的病历的查找概率必然高于健康学生病历的查找概率。在不等概率的情况下，顺序查找的平均查找长度为：

$$ASL'_{sq}=p_1+2p_2+\cdots+(n-1)p_{n-1}+np_n$$

显然，当 $p_n\leqslant p_{n-1}\leqslant\cdots\leqslant p_1$ 时 ASL'_{sq} 达到最小值。因此，若事先知道表中各节点的查找概率不相等和它们的分布情况，则应将表中节点按查找概率由大到小的顺序存放，以便提高顺序查找的效率。然而，在一般情况下，各节点的查找概率无法事先确定。为了提高查找效率，我们可以对算法 SeqSearch 做如下修改：每当查找成功时，就将找到的节点和其后继节点(若存在)交换。这样，使得查找概率大的节点在查找过程中不断往后移，便于在以后的查找中减少比较次数。

顺序查找的优点是算法简单，且对表的结构无任何要求，无论是用数组还是用链表来存放节点，也无论节点之间是否按关键字有序或无序，它都同样适用。顺序查找的缺点是查找效率较低，特别是当 n 较大时，不宜采用顺序查找。

9.2.2　二分查找

二分查找(Binary Search)又称折半查找，它是一种效率较高的查找方法。但是，二分查找有一定的条件限制：要求线性表必须采用顺序存储结构，而且表中元素必须按关键字有序(升序或降序均可)，即要求线性表是有序表。在下面的讨论中，不妨设有序表是递增有序的。

二分查找的基本思想：首先将待查的 K 值和有序表 $R[0]$到 $R[n-1]$的中间位置 mid 上的节点的关键字进行比较，若相等，则查找完成，返回此位置的值；若 $R[mid].key>K$，则说明待查找的节点只可能在左子表 $R[0]$到 $R[mid-1]$中，下面只要在左子表中继续进行二分查找，若

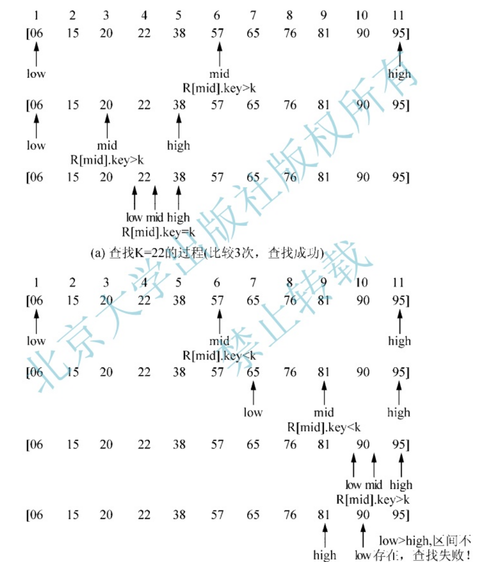
R[mid].key<K，则说明待查找的节点只可能在右子表 R[mid+1]到 R[n−1]中，因此要在右子表中继续进行二分查找。这样，经过一次关键字比较就缩小一半的查找区间。不断重复上述查找过程，直到找到关键字为 K 的节点，或者当前的查找区间为空(表示查找失败)。

如果分别用 low 和 high 来表示当前查找区间的下界(第一个记录的位置)和上界(最后一个记录的位置)，则该区间的中点位置 $\mathrm{mid}=\frac{\mathrm{low}+\mathrm{high}}{2}$。

例如，假设被查找的有序表中关键字序列为：

06，15，20，22，38，57，65，76，81，90，95

当给定的 K 值分别为 22 和 85 时，进行二分查找的过程如图 9.1 所示。



(a) 查找K=22的过程(比较3次，查找成功)

(b) 查找K=85的过程(比较3次后查找失败)

图 9.1 二分查找过程示例

二分查找算法的代码实现如下。

算法 9.2 二分查找算法。

```
int BinSearch(SeqList R,KeyType K)
/* 在有序表 R 中进行二分查找,成功时返回节点的位置,失败时返回-1 */
```

```
{
int low, mid, high;
    low=0; high=n-1;          /* 置查找区间的下、上界初值 */
    while(low<=high)          /* 当前查找区间非空 */
    {
        mid=(low+high)/2;
        if(K= =R[mid].key)
            return mid;       /* 查找成功返回 */
        if(K<R[mid].key)
            high=mid-1;       /* 缩小查找区间为左子表 */
        else  low=mid+1;      /* 缩小查找区间为右子表 */
    }
    return(-1) ;              /* 查找失败 */
}
```

二分查找过程可用二叉树来描述，把当前查找区间的中间位置 mid 上的节点作为根，左半区间和右半区间中的节点分别作为根的左子树和右子树，左半区间和右半区间再按类似的方法类推，由此得到的二叉树，称为描述二分查找的判定树(Decision Tree)。

例如，上述具有 11 个节点的有序表可以用图 9.2 所示的判定树表示，树中节点内的数字表示该节点在有序表中的位置。若查找的节点是表中第 6 个节点，则只需进行一次比较；若查找的节点是表中第 3 或第 9 个节点，则需进行两次比较；查找第 1、4、7、10 个节点需要比较三次，查找第 2、5、8、11 个节点需要比较四次。由此可见，二分查找过程恰好是一条从判定树的根到被查节点的路径，经历比较的关键字个数恰为该节点在树中的层数。若查找失败，则其比较过程是一条从判定树根到某个外部节点(非表中节点)的路径，所需的关键字比较次数是该路径上内部节点的个数。

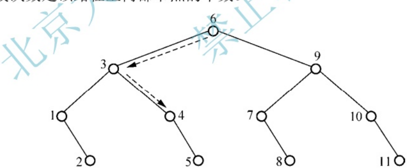


图 9.2 具有 11 个节点时二分查找的判定树

用图 9.2 所示判定树描述图 9.1(a)查找 K=22 的过程时，所经历的比较路径如图 9.2 中虚线所示，查找过程将 K 分别与节点 6、3 和 4(即关键字 57、20 和 22)比较，共进行了三次比较后才查找成功。要查找 K=85 的记录，所经过的内部节点为 6、9、10，最后达到外部节点，表明查找失败，其比较次数为三次。

借助于二分查找判定树，我们很容易求得二分查找的平均查找长度。设节点总数 $n=2^h-1$，则判定树是深度为 $h=\log_2(n+1)$的满二叉树，树中第 k 层上的节点个数为 2^{k-1}，查找它们所需的比较次数是 k，因此，在等概率假设下，二分查找的平均查找长度为：

$$\mathrm{ASL_{bn}}=\sum_{i=1}^{n} p_i c_i=\sum_{i=1}^{n} c_i/n=\sum_{k=1}^{n} k\times 2^{k-1}/n=((n+1)*\log_2(n+1)-1)/n \qquad (9.3)$$

当 n 很大时，可使用近似公式：

$$ASL_{bn}=\log_2(n+1)-1 \tag{9.4}$$

作为二分查找的平均查找长度。二分查找在查找失败时所需比较的关键字个数不超过判定树的深度，在最坏情况下查找成功的比较次数也不超过判定树的深度，因为判定树中度数小于 2 的节点可能在最下面的两层上，所以 n 个节点的判定树的深度和 n 个节点的完全二叉树相同，均为 $\lceil \log_2(n+1) \rceil$。由此可见，二分查找的最坏性能和平均性能相当接近。

虽然二分查找的效率高，但是要将表按关键字排序。而排序本身是一种很费时的运算，即使采用高效率的排序方法也要花费 $O(n\log_2 n)$的时间。而且二分查找只适用于顺序存储结构，为了保持表的有序性，在顺序结构里插入和删除都必须移动大量的节点。因此，二分查找特别适用于那种一经建立就很少改动、而又经常需要查找的线性表。对那些查找少又经常需要改动的线性表，可采用链表作为存储结构，进行顺序查找。

9.2.3 分块查找

分块查找(Blocking Search)又称索引顺序查找，它是一种性能介于顺序查找和二分查找之间的查找方法。它要求按索引的方式来存储线性表，即分块查找表由“分块有序”的线性表和索引表组成。

(1) “分块有序”的线性表：将表 $R[n]$均分为 b 块，前 b−1 块中节点个数为 $S=\lceil n/b \rceil$，第 b 块的节点数小于等于 S；每一块中的关键字不一定有序，但前一块中的最大关键字必须小于后一块中的最小关键字，即表是“分块有序”的。

(2) 索引表：抽取各块中的最大关键字及其起始位置构成一个索引表 ID[b]，即 ID[i]($0 \leqslant i < b$)中存放着第 i 块的最大关键字及该块在表 R 中的起始位置。由于表 R 是分块有序的，所以索引表是一个递增有序表。

图 9.3 就是满足上述存储要求的分块查找表，其中 R 只有 18 个节点，被分成三块，每块中有六个节点，第一块中最大关键字 22 小于第二块中最小关键字 24，第二块中最大关键字 48 小于第三块中最小关键字 49。

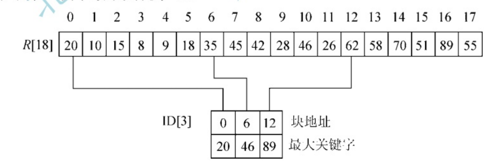


图 9.3 分块有序表的索引存储表示

分块查找的基本思想：先查找索引表，因为索引表是有序表，故可采用二分查找或顺序查找，以确定待查的节点在哪一块；然后在已确定的那一块中进行顺序查找。

在如图 9.3 所示的存储结构中，查找关键字等于给定值 K=28 的节点，因为索引表小，不妨用顺序查找方法查找索引表，即首先将 K 依次和索引表中各关键字比较，直到找到第一个关键字大于等于定值 K 的节点，由于 K>20，继续比较下一个值，K<46，因此可以判定关键字为 28 的节点若存在，则必定在第二块中；然后，由 ID[1].addr 找到第二块的起始地址 7，从该地址开始进行顺序查找，直到 R[9].key=K 为止。若给定值 K=30，类似地，

通过索引表先确定在第二块，然后在该块中查找，若查找不成功，则说明表中不存在关键字为 30 的节点。

当选用二分查找法查找索引表时，分块查找算法及有关说明如下。

```
typedef struct      /* 索引表的节点类型 */
{
  KeyType key;
  int addr;
} IDtable;
IDtable ID[b];      /* 索引表 */
```

算法 9.3　分块查找算法。

```
int BlkSearch(SeqList R,IDtable ID[b],KeyType K)
{// 分块查找,成功时函数值为关键字等于 K 的节点在 R 中的序号,失败时函数值为-1
    int i, low1 ,low2, mid, high1, high2;
    low1=0;  high1=b-1;       /*设置二分查找区间下、上界的初值 */
    while (low1<=high1)
    {
        mid=(low1+high1)/2;
        if(K<=ID[mid].key)  high1=mid-1;
        else  low1=mid+1;
    }                         /* 查找完毕,low1 为找到的块号 */
    if(low1<b)                /*若 low1=b,则 K 大于 R 中的所有关键字*/
{
        low2=ID[low1].addr;  /* 块起始地址 */
        if(low1==b-1)  high2=n-1;        /* 求块未地址 */
        else high2=ID[low1+1].addr-1;
        for (i=low2; i<=high2; i++)      / * 在块内顺序查找 */
            if (R[i].key==K)  return i;  /* 查找成功 */
        return (-1);                     /* 查找失败 */
    }
}
```


由于分块查找实际上是两次查找过程，故整个算法的平均查找长度是两次查找的平均查找长度之和。

(1) 以二分查找来确定块，则分块查找的平均查找长度为：

$$ASL_{blk}=ASL_{bn}+ASL_{sq}\approx\log_2(b+1)-1+(s+1)/2\approx\log_2(n/s+1)+s/2$$

(2) 以顺序查找确定块，则分块查找的平均查找长度为：

$$ASL'_{blk}=(b+1)/2+(s+1)/2-(s^2+2s+n)/(2s)$$

当 $s=\sqrt{n}$ 时 ASL'_{blk} 取极小值 $\sqrt{n}+1$，即当采用顺序查找确定块时，应将各块中的节点数选定为 n。例如，若表中有 10000 个节点，则应把它分成 100 个块，每块中含 100 个节点。用顺序查找确定块，分块查找平均需要做 100 次比较，而顺序查找平均需做 5000 次比

较，二分查找最多需 14 次比较。由此可见，分块查找算法的效率介于顺序查找和二分查找之间。

在实际应用中，分块查找不一定要将线性表分成大小相等的若干块，而应该根据表的特征进行分块。例如，一个学校的学生登记表，可按系号或班号分块，此外，各块中的节点也不一定要存放在同一个数组中，可将各块放在不同的数组中，也可将每一块存放在一个单链表中。

分块查找的优点：在表中插入或删除一个记录时，只要找到该记录所属的块，即可在该块内进行插入和删除运算，而且对表中数据进行插入或删除比较容易，无需移动大量记录，因为块内记录的存放是任意的。分块查找的主要代价是增加了一个辅助数组的存储空间和将初始表分块排序的运算。

9.3 哈希表查找

在前面讨论的表示查找表的各种结构的共同特点是记录在表中的位置和它的关键字之间不存在确定的关系，查找的过程为给定值依次和关键字集合中各个关键字进行比较。查找的效率取决于和给定值进行比较的关键字次数。用这类方法表示的查找表，其平均查找长度都不为 0。不同的表示方法的差别在于：关键字和给定值进行比较的顺序不同。对于频繁使用的查找表，如果希望 ASL=0，就需要预先知道所查关键字在表中的位置，即要求记录在表中位置和其关键字之间存在一种确定的关系。这就是本节介绍的哈希表查找方法的基本思想。

9.3.1 哈希表的概念

哈希法(Hashing)是一种重要的存储方法，也是一种常见的查找方法。它的基本思想如下：以节点的关键字 K 为自变量，通过一个确定的函数关系 f，计算出对应的函数值 $f(K)$，把这个值解释为节点的存储地址，将节点存入 $f(K)$所指的存储位置。查找时再根据要查找的关键字用同样的函数计算地址，然后到相应的单元里去取要找的节点。因此，哈希法又称关键字——地址转换法。用哈希法存储的线性表称为哈希表(Hash Table)，上述的函数 f 称为哈希函数，$f(K)$称为哈希地址。

通常哈希表的存储空间是一个一维数组，哈希地址是数组的下标，在不致引起混淆的情况下，这个一维数组空间简称为哈希表。

假如要建立一张全国各城市的人口统计表，如表 9.1 所示。

表 9.1 全国人口统计表

编号	城市名	省/直辖市	非农业人口/万	总人口/万
1	北京	北京市	904	1145
2	上海	上海市	1174	1309
3	杭州	浙江省	269	419

显然，可以按编号依次存放这张表，编号就是记录的关键字，由它唯一确定记录的存储位置，如北京编号为 1，要查看北京的人口，只要取出第 1 条记录即可。如果把这个存储方式看作哈希表，则哈希函数 f(key)=key，有 f(1)=1，f(2)=2，…，为了查看方便，也可以用地区名做关键字，取地区名的第一个拼音字母的序号做哈希函数，则有 f(beijing)=2，f(chengdu)=3，f(shanghai)=19。

从这个例子可以看出，哈希函数是一个映像，即将关键字的集合映射到某个地址集合上，它的设置很灵活，只要这个地址集合的大小不超出允许范围即可；由于关键字的值域往往比哈希表的个数大很多，所以哈希函数是一个压缩映像。例如，对于如下九个关键字{Zhao，Qian，Sun，Li，Wu，Chen，Han，Ye，Dai}，设哈希函数：

$f(\text{key})=\lfloor (\text{ord}(\text{第一个字符})-\text{ord}('A')+1)/2 \rfloor$，ord 为字符的次序，如 ord(‘A’)=1，则构建的哈希表如表 9.2 所示。

表 9.2　哈希表

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	Chen	Dai		Han		Li		Qian	Sun		Wu	Ye	Zhao

如果要查找给定关键字为“Qian”的记录，则按上述哈希函数进行计算，得到 f(Qian)=8，即可从地址为 8 的表中取得该记录。但是，当同时存在关键字“Zhao”和“Zhang”时，得到 f(Zhao)=13=f(Zhang)，这时就产生了“冲突”。一般来讲，很难找到一个不产生冲突的哈希函数，只能根据实际情况选择恰当的哈希函数，使冲突尽可能少产生。因此，在构造这种特殊的“查找表”时，除了需要选择一个“好”(尽可能少产生冲突)的哈希函数之外，还需要找到一种“处理冲突“的方法。

所以哈希表可以定义如下：根据设定的哈希函数 f(key)和所选中的处理冲突的方法，将一组关键字映像到一个有限的、地址连续的地址集(区间)上，并以关键字在地址集中的“像”作为相应记录在表中的存储位置，如此构造所得的查找表称为“哈希表”，这一映像过程也称为“散列”，所以哈希表也称为散列表。

9.3.2　哈希表的构造

哈希函数的种类繁多，这里不能一一列举。下面仅介绍几种计算简单且效果较好的哈希函数。怎样什么才算是好的哈希函数呢？有两个原则可以参考：计算简单，如果一个算法可以保证所有的关键字都不会产生冲突，但是这个算法需要很复杂的计算，会耗费很多时间，那么对于需要频繁查找的表来说，就会大大降低查找的效率，因此哈希函数的计算时间不应该超过其他查找技术与关键字比较的时间；哈希地址分布均匀，即尽量让哈希地址均匀分布在存储空间中，保证存储空间的有效利用，并减少为处理冲突而耗费的时间。

为了方便讨论，以下均假定关键字是数字型的，若关键字是字符型的，则可先将其转换成数值。

1. 直接定址法

如果现在要统计 0～100 岁的人口数字，如表 9.3 所示，那么对年龄这个关键字就可以直接用年龄的数字作为地址，即 f(key)=key。若现在要统计的是 1980 年后出生的人口数，

如表 9.4 所示，那么对于出生年份这个关键字可以用年份减去 1980 来作为地址，此时 $f(\text{key})=\text{key}-1980$。也就是说，可以取关键字的某个线性函数值为哈希地址，即

表 9.3　人口数字统计表

地址	年龄	人数/万
00	0	500
01	1	600
02	2	450
…	…	…
20	20	1500
…	…	…

表 9.4　1980 年后出生的人口统计

地址	出生年份	人数/万
00	1980	1500
01	1981	1600
02	1982	1450
…	…	…
20	2000	800
…	…	…

$f(\text{key})=a\times\text{key}+b$　　(a、b 为常数)

这样的哈希函数优点就是简单、均匀，也不会产生冲突，但是却需要事先知道关键字的分布情况，适合查找表较小且连续的情况。由于这样的限制，在现实应用中，此方法虽然简单，却不常用。

2. 数字分析法

该方法是提取关键字中取值较均匀的数字作为哈希地址的方法。它适合于所有关键字已知的情况，并需要对关键字中每一位的取值分布情况进行分析。例如，一组关键字为{87912602，87956671，87937615，8784675}，分析可知，每个关键字从左到右的第 1、2、3 位和第 6 位取值比较集中，不宜作为哈希函数，剩余的 4、5、7、8 位取值比较分散，可根据实际需要取其中的若干位作为哈希地址。若取最后两位作为哈希地址，则哈希地址为{2，71，15，75}。

数字分析法通常适合处理关键字位数比较大的情况，如果事先知道关键字的分布且关键字的若干位分布较均匀，就可以考虑使用这种方法。

3. 平方取中法

通常，要预先估计关键字的数字分布并不容易，要找数字均匀分布的位数则更难。例如，一组关键字(0l00，0110.1010，1001，0111)就无法使用数字选择法得到较均匀的散列函数。此时可采用平方取中法，即先通过求关键字的平方值扩大差别，然后取中间的几位或其组合作为哈希地址。因为一个乘积的中间几位数和乘数的每一位都相关，故由此产生的哈希地址也较为均匀，所取位数由哈希表的表长决定。

例如，上述一组关键字的平方结果是(0010000，0012100，1020100，1002001，0012321)，



若表长为 1000，则可取中间三位作为哈希地址集：

(100，121，201 020，123)

平方取中法比较适合不知道关键字的分布，而位数又不是很大的情况。

4. 折叠法

折叠法是将关键字分割成位数相同的几部分(最后一部分的位数可以不同)，然后取它们的叠加和(舍去进位)为哈希地址的方法。折叠法又分移位叠加法和边界叠加法两种。移位叠加法将分割后的几部分最低位对齐，然后相加；边界叠加法则从一端沿着分割边界来回折叠，然后对齐相加。例如，关键字 key=0442205864，按移位叠加法和边界叠加法计算哈希地址，如图 9.4 所示。将此关键字分成四位一段，两种叠加结果如下。

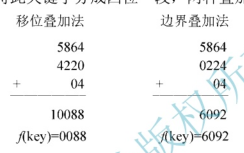


图 9.4　折叠法示例

折叠法实现不需要知道关键字的分布，适合关键字位数比较多，且每一位数字的分布基本均匀的情况。

5. 除留余数法

选择一个不大于哈希表长 m 的正整数 P，用 P 去除关键字后所得的余数作为哈希地址，即

$$H(\text{key})=\text{key}\%P \tag{9.5}$$

该方法计算简单，适用范围大，是一种最常用的构造哈希函数的方法，其关键是选取适当的 P，如果选择不当，容易产生较多同义词，使哈希表中有较多的冲突。

如表 9.5 所示，对有 12 个记录的关键字构造哈希表时，用 f(key)=key%12 的方法，如 29%12=5，所以它存储在下标为 5 的位置。

表 9.5　除留余数法构造哈希表

下标	0	1	2	3	4	5	6	7	8	9	10	11
关键字	12	25	38	15	16	29	78	67	56	21	22	47

不过这也有存在冲突的可能，如果关键字中有像 18、30、42 等数字，它们的余数都是 6，就和 78 所对应的下标位置冲突了。

因此，根据理论研究，若哈希表表长为 m，通常 P 为小于或等于表长(最好接近 m)的最小质数或不包含小于 20 质因子的合数。

6. 随机数法

选择一个随机函数，取关键字的随机函数值为它的散列地址，即

$$f(\text{key})=\text{random (key)} \tag{9.6}$$

这里的 random 为随机函数。通常，当关键字长度不等时采用此法构造散列地址较恰当。

实际工作中需视不同的情况采用不同的哈希函数。通常需考虑的因素有计算哈希函数所需的时间(包括硬件指令的因素)、关键字的长度、哈希表的大小、关键字的分布情况和记录的查找频率。

9.3.3 解决冲突的方法

在哈希表中，虽然冲突很难避免，但是发生冲突的可能性却有大有小。这主要和三个因素有关。

(1) 与装填因子有关。装填因子是指哈希表中已存入的记录数 n 与哈希地址空间大小 m 的比值，即 $a=n/m$。a 越小，冲突的可能性就越小；a 越大(最大可能取 1)，冲突的可能性就越大。因为 a 越小，哈希表中的空闲单元的比例就越大，所以待插入记录同已插入记录发生冲突的可能性就越小；反之，哈希表中的空闲单元的比例就越小，待插入记录同已插入记录发生冲突的可能性就越大。此外， a 越小，存储空间的利用率就越低，反之利用率就越高。为了兼顾减少冲突的发生和提高存储空间的利用率，通常最终使 a 控制在 0.6～0.9。

(2) 与所采用的哈希函数有关。若哈希函数选择得当，就可以使哈希地址尽可能均匀地分布在哈希地址空间中，从而减少冲突的发生。否则，就可能使哈希地址集中于某些区域，从而加大冲突的产生。

(3) 与解决冲突的哈希冲突函数有关。哈希冲突函数的选择的好坏也会减少或增加发生冲突的可能性。

下面介绍处理哈希冲突两大类方法：开放地址法和拉链法。

1. 开放地址法

用开放地址法解决冲突的做法：当冲突发生时，使用某种方法在哈希表中形成一个探查序列，沿着此探查序列逐个单元地查找，直到找到给定的关键字或者碰到一个开放的地址(即该地址单元为空)为止。若插入时碰到开放的地址，则可将待插入新节点存放在该地址单元中。若查找时碰到开放的地址，则说明表中没有待查的关键字。显然，用开放地址法建立哈希表，建表前必须将表空间的所有单元置空。

形成探查序列的方法不同，所得到的解决冲突的方法也不相同。下面介绍几种常用的探查方法，并假设哈希表 HT 的长度为 m，节点个数为 n。

1) 线性探查法

线性探查法的基本思想：将哈希表看作一个环形表。若地址为 d(即 H(key)=d)的单元发生冲突，则依次探查下述地址单元：

$$d+1，d+2，\cdots，m-1，0，1，\cdots，d-1$$

直到找到一个空单元或查找到关键字为 key 的节点为止。当然，若沿着该探查序列查找一遍之后，又回到了地址 d，则无论是做插入操作还是做查找操作，都意味着失败(即此时表满)。

用线性探查法解决冲突，求下一个开放地址的公式为：

$$d_i=(d+i)\%m \qquad i=l，2，\cdots，s(1\leqslant s\leqslant m-1) \tag{9.7}$$

式中，$d=f(\text{key})$。

例如，已知一组关键字为(26，36，41，38，44，15，68，12，06，51，25)，用线性

探查法解决冲突构造这组关键字的哈希表。

为了减少冲突，通常令装填因子＜1。在此，我们取 0.75。因为 n=11，所以，哈希表长 m=15，即哈希表为 HT[15]。利用除余法构造哈希函数，选 P=13，即哈希函数为 f(key)=key%13。

插入时，首先用哈希函数计算出散列地址 d，若该地址是开放的，则插入新节点；否则用式(9.7)求下一个开放地址。第一个插入的是 26，它的哈希地址 d 为 f(26)=26%13=0，因为这是一个开放地址，故将 26 插入 HT[0]。类似地，依次插入 36，41，38 和 44 时，它们的哈希地址 10、2、12、5 都是开放的，故将它们分别插入 HT[10]、HT[2]、HT[12]和 HT[5]中。当插入 15 时，其哈希地址为 d=f(15)=2，由于 HT[2]已被关键字 41 占用(即发生冲突)，故利用式(9.1)进行探查。显然，d_1=(2+1)%15=3 为开放地址，因此，将 15 插入 HT[3]中。类似地，68 和 12 均经过一次探查后，分别插入到 HT[4]和 HT[3]中。06 直接插入 HT[6]，51 的哈希地址为 12，与 HT[12]中的 38 发生冲突，故由式(9.7)求得 d_1=13，仍然冲突，再次探查下一个地址 d_2=14，该地址是开放的，故将 51 插入 HT[14]；最后一个插入的是 25，它的哈希地址也是 12，经过了四次探查 d_1=13，d_2=14，d_3=0，d_4=1 之后才找到开放地址 1，将 25 插入 HT[1]。由此构造的哈希表如表 9.6 所示，其中最末一行的数字表示查找该节点时所进行的关键字比较次数。

表 9.6　用线性探查法构造散列表示例

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
关键字	26	25	41	15	68	44	06				36		38	12	51
比较次数	1	5	1	2	2	1	1				1		1	2	3

在上例中，f(15)=2，f(68)=3，即 15 和 68 不是同义词，但由于处理 15 和同义词 41 的冲突时，15 抢先占用了 HT[3]，这就使得插入 68 时，这两个本来不应该发生冲突的非同义词之间也会发生冲突。一般地，用线性探查法解决冲突时，当表中 i，i+1，…，i+k 位置上已有节点时，一个哈希地址为 i，i+1，…，i+k+1 的节点都将插入位置 i+k+1 上，我们把这种哈希地址不同的节点，争夺同一个后继哈希地址的现象称为“堆积”。这将造成不是同义词的节点，处在同一个探查序列之中，从而增加了探查序列的长度。若哈希函数选择不当或装填因子过大，都可能使堆积的机会增加。

为了减少堆积的机会，就不能像线性探查法那样探查一个顺序的地址序列，而应该使探查序列跳跃式地散列在整个哈希表中。为此下面介绍另外三种解决冲突的方法，与线性探查法相比，它们大大地减少了堆积的可能性。

2) 二次探查法

二次探查法的探查序列依次是 1^2，-1^2，2^2，-2^2，…，也就是说，发生冲突时，将同义词来回散列在第一个地址 d=f(key)的两端。由此可知，发生冲突时，求下一个开放地址的公式为：

$$D_{2i-1}=(d+i^2)\%m$$

$$D_{2i}=(d-i)\%m \qquad (1\leqslant i\leqslant (m-1)/2) \tag{9.8}$$

虽然二次探查法减少了堆积的可能性，但是二次探查法不容易探查到整个哈希表空间。

3) 随机探查法

采用随机探查法解决冲突时，求下一个开放地址的公式为：

$$d_i=(d+R_i)\%m \qquad (1\leqslant i\leqslant m-1) \qquad (9.9)$$

式中，$d=f(key)$，R_1，R_2，…，R_{m-1} 是 1，2，…，$m-l$ 的一个随机排列。如何得到随机排列，涉及随机数的产生问题。在实用中，常常用移位寄存器序列代替随机数序列。设 m 是 2 的方幂，K 是 1～m-1 的一个整数，产生移位寄存器序列的方法如下。

任取 1～m-1 的一个整数作为 R_1。

设已知 R_{i-1}，令

$$R_i\begin{cases}2R_{i-1} & \text{当}2R_{i-1}<m\text{时}\\(2R_{i-1}-m)\oplus K & \text{当}2R_{i-1}\geqslant m\text{时}\end{cases}$$

式中：整数 K、m、R_{i-1}、R_i 都是二进制表示，运算 ⊕ 为按位模 2 加法，按位模 2 加法与普通二进制加法类似，只是不产生进位。

应当注意，K 必须选择合适才能产生出 1，2，…，m-1 的一个随机排列。例如，设 m=8，取 K=3，R_i=5，则产生的随机排列为 5，1，2，4，3，6，7。若取 K=5，也能产生 1～7 的一个随机排列，但 K 取其他值就不能产生了。

4) 双重哈希函数探查法

这种方法使用两个哈希函数 H_1 和 H_2，其中 H_1 和前面的 H 一样，以关键字为自变量，产生一个 0 到 m-1 之间的数作为哈希地址，H_2 也以关键字为自变量，产生一个 1 到 m-1 之间的并和 m 互素的数作为对哈希地址的补偿。若 H_1(key)=d 时发生冲突，则再计算 H_2(key)，得到的探查序列为：

$$(d+H_2(\text{key}))\%m,\quad (d+2H_2(\text{key}))\%m,\quad (d+3H_2(\text{key}))\%m,\ \cdots$$

由此可知，双哈希函数探查法求下一个开放地址的公式为：

$$d_i=(d+iH_2(\text{key}))\%m \qquad (1\leqslant i\leqslant m-1) \qquad (9.4)$$

定义 H_2(key)的方法较多，但无论采用什么方法定义 H_2，都必须使 H_2(key)的值和 m 互素，才能使发生冲突的同义词地址均匀地分布在整个表中，否则可能造成同义词地址的循环计算。

若 m 为素数，则 H_2(key)取 1～m-1 的任何数均与 m 互素，因此，我们可以简单地将 H^2 定义为：

$$H_2(\text{key})=\text{key}\%(m-2)+1$$

若 m 是 2 的方幂，则 H_2(key)可取 1～m-1 之间的任何奇数。

2. 拉链法

拉链法是将所有具有相同哈希地址的关键字的值放在同一个单链表中。若选定的哈希表的长度为 m，则可将散列表定义为一个由 m 个头指针组成的指针数组 $T[m]$，凡是哈希地址为 i 的节点，均插入到以 $T[i]$为头指针的单链表中。T 中各分量的初值为空。

例如，给定关键字集合{26，36，41，38，44，15，68，12，06，51}，取哈希表长 m=13，哈希函数为 f(key)=key%13，用拉链法解决冲突所构造出来的哈希表，如图 9.5 所示。

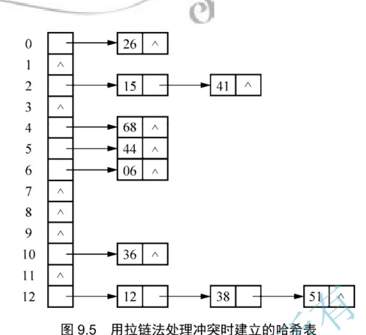


图 9.5 用拉链法处理冲突时建立的哈希表

与开放地址法相比，拉链法有如下几个优点：拉链法不会产生堆积现象，因此平均查找长度较短；由于拉链法中各单链表上的节点空间是动态申请的，故它更适合于构造表前无法确定表长的情况；在用拉链法构造的哈希表中，删除节点的操作易于实现，只要简单地删去链表上相应的节点即可。而对开放地址法构造的哈希表，删除节点不能简单地将被删节点的空间置为空，否则将截断在它之后填入哈希表的同义词节点的查找路径，这是因为在各种开放地址法中，空地址单元(即开放地址)都是查找失败的条件。因此在用开放地址法处理冲突的散列表上执行删除操作，只能在被删节点上做删除标记，而不能真正删除节点。

当装填因子较大时，拉链法所用的空间比开放地址法多，但是空间越大，开放地址法所需的探查次数越大，所以拉链法所增加的空间开销是合算的。

9.3.4 哈希表查找实现

哈希表的查找过程和建表过程相一致。假设给定的值为 key，根据建表时设定的哈希函数 *f*，计算出哈希地址 *f*(key)，若该地址对应的空间为空闲，则查找失败，否则将该地址中的节点与给定值 key 进行比较，若值相等则查找成功，否则按建表时设定的处理冲突方法求下一个地址，反复进行该步骤，直到某个地址空间为空闲(查找失败)或者关键字比较相等(查找成功)为止。

下面以线性探查法为例，给出哈希表上的查找和插入算法。

哈希表类型说明如下。

```
 #define NIL -1
 #define M 997
 typedef struct{
    KeyType key;
    InfoType otherinfo;
 }NodeType;
Typedef  NodeType  HashTable[m];
```

算法 9.4　哈希表的查找。

```
int Hash(Keytype k,int i)
{ //散列函数用除余法构造,并使用线性探查的开放定址法处理冲突
    return(k/m+i)%m;
}

int HashSearch(HashTable T,Keytype K,int *pos)
{ //开放定址法的哈希表查找算法
    int i=0;
    do{
        *pos=Hash(K,i);                    /*求探查地址 hi*/
        if(T[*pos].key==K) return 1;       /*查找成功返回*/
        if(T[*pos].key==NIL) return 0;     /*查找到空节点返回*/
      }while(++i<m)                        /*最多做 m 次探查,m 是表长*/
    return -1;                             /*表满且未找到时,查找失败*/
}

void Hashlnsert(HashTable T,NodeTypene w)
{/*将新节点 new 插入哈希表 T[0..m-1]中*/
    int pos,sign;
    sign=HashSearch(T,new.key,&pos);/*在表 T 中查找 new 的插入位置
    if(!sign)                              /*找到一个开放的地址 pos*/
        T[*pos]=new;                       /*插入新节点 new,插入成功*/
    else                                   /*插入失败*/
        if(sign>0)
            printf("duplicate key!");      /* 有重复的关键字*/
        else
            Error("hashtableoverflow!");/*表满错误,终止程序执行*/
}

void CrHTable(HashTable T,NodeType A[ ] ,int n)
{     /*根据 A[0…n-1]中节点建立哈希表 T[0…m-1] */
    int i;
    if(n>m) /*用开放定址法处理冲突,装填因子(a=n/m)<1*/
        Error("Load factor>1");
    for(i=0;i<m;i++)
        T[i].key=NULL;         /*将各关键字清空,使地址 i 为开放地址*/
    for(i=0;i<n;i++)           /*依次将 A[0…n-1]插入到哈希表 T[0…m-1]中*/
        Hashlnsert(T,A[i]);  /*调用插入算法,将 A[i]插入哈希表 T 中*/
}
```

从上述查找过程可知，虽然哈希表在关键字和存储位置之间直接建立了对应关系，但是，由于冲突的产生，哈希表的查找过程仍然是一个和关键字比较的过程，不过哈希表的平均查找长度比顺序查找要小得多，比二分查找长度也小。

9.4 编程实现电话号码查询系统

采用哈希表实现电话号码查找系统，并考虑解决哈希冲突的方法。设每个记录有电话号码、用户名和地址三个数据项，需要建立哈希表存储记录。为插入查找方便，在创建哈希表时建立了两个哈希表，一个以电话号码作为关键字建立哈希表，另一个以用户名为关键字建立哈希表。

由于存储结构是链表，因此采用拉链法解决哈希冲突问题。定义 NUM 型节点，包含电话号码 num，姓名 name，地址 address 和指向下一个节点的指针，同义词节点组成一个链表，其中所有链表地址又组成一个数组。

(1) 数据类型定义如下。

```
typedef struct node
{
    char num[11],name[15],address[20];  //电话号码,用户名,地址
    struct node *next;                  //下一个节点指针
}NUM,NAME;
NUM* num_list[19];         //电话号码为存储关键字的哈希表中存储链表的头指针数组
NAME* name_list[19];       //用户名为存储关键字的哈希表中存储链表的头指针数组
```


(2) 主程序代码如下。

```
int hash1(char num[]) /*电话号码为关键字用除余法求哈希地址,输入的号码转换为整数进行除余运算*/
{
        int i,k=0;
        for(i=0;num[i]!='\0';i++)
        {
            k=10*k+num[i]-48;
        }
        k=k%19;
        return k;
}

int hash2(char name[])//用户名为关键字,用除余法求哈希地址
{
        int i,k=0;
        for(i=0;name[i]!='\0';i++)
        {
            k=10*k+name[i];
        }
        k=k%19;
        return k;
```

```
}

void create()//创建两个哈希表,一个以电话号码为关键字,一个以姓名为关键字
{
        char num[11],name[15],address[20];
        struct node *p1;
        struct node *q1;
        int k1,k2;
        printf("请输入信息:电话号码  姓名  地址,以# # #结束\n");
        scanf("%s%s%s",num,name,address);
        while(strcmp(num,"#")!=0)
        {
            p1=(struct node*)malloc(sizeof(struct node));/*生成节点p,插
入到号码所在哈希表*/
            q1=(struct node*)malloc(sizeof(struct node));/*生成节点q,插
入到姓名所在哈希表*/
            strcpy(p1->num,num);
            strcpy(p1->name,name);
            strcpy(p1->address,address);
            k1=hash1(num);
            p1->next=num_list[k1];
            num_list[k1]=p1;
            strcpy(q1->num,num);
            strcpy(q1->name,name);
            strcpy(q1->address,address);
            k2=hash2(name);
            q1->next=name_list[k2];
            name_list[k2]=q1;
            scanf("%s%s%s",num,name,address);
        }
        printf("哈希表创建成功! \n");
}

void print() /*输出哈希表,可以选择输出电话号码为关键字的链表或名称为关键字的哈希表*/
{
    struct node *f;
    int x,i;
    printf("请输入你的选择: \n");
    printf("1.显示电话号码表");
    printf("2.显示姓名表");
    scanf("%d",&x);
    switch(x)
    {
```

```
                case 1:for(i=0;i<19;i++)
                    {
                        printf("%d:",i);
                        f=num_list[i];
                        while(f!=NULL)
                        {
                         printf("--> 号 码 :%s  姓 名 :%s  地 址 :%s",f->num,f->name,
f->address);
                         f=f->next;
                        }
                        printf("\n");
                    }
                    break;
                case 2:for(i=0;i<19;i++)
                    {
                        printf("%d:",i);
                        f=name_list[i];
                        while(f!=NULL)
                        {
                            printf("-->号码:%s 姓名:%s 地址:%s",f->num,f->name,
f->address);
                            f=f->next;
                        }
                        printf("\n");
                    }
                    break;
            }
        }

        void insert()  //插入节点,输入一个节点信息
        {//分别在电话号码为关键字和名称为关键字的哈希表中各插入一个节点
            char num[11],name[15],address[20];
            struct node *p1;
            struct node *q1;
            int k1,k2;
            printf("请输入新节点的信息:号码   姓名   地址\n");
            scanf("%s%s%s",num,name,address);
            p1=(struct node*)malloc(sizeof(struct node));
            q1=(struct node*)malloc(sizeof(struct node));
            strcpy(p1->num,num);
            strcpy(p1->name,name);
            strcpy(p1->address,address);
            k1=hash1(num);
```

```
        p1->next=num_list[k1];
        num_list[k1]=p1;
        strcpy(q1->num,p1->num);
        strcpy(q1->name,p1->name);
        strcpy(q1->address,p1->address);
        k2=hash2(q1->name);
        q1->next=name list[k2];
        name_list[k2]=q1;
        printf("OK\n");
    }

    void search()  //链表查询,可以根据号码和姓名两种方式查询
    {
        char num[11],name[15];
        int k1,k2,x,find=0;
        struct node *f;
        printf("请输入查找选项: \n");
        printf("1.根据电话号码查询\n");
        printf("2.请根据姓名查询\n");
        scanf("%d",&x);
        switch(x)
        {
            case 1: printf("请输入电话号码: ");
            scanf("%s",num);
            k1=hash1(num);
            f=num_list[k1];
            while (f!=NULL)
            {
                if(strcmp(f->num,num)==0)
                {
                    printf("找到信息:号码-%s 姓名-%s 地址-%s\n",f->num,
f->name,f->address);
                 find=1;
                }
                f=f->next;
            }
            if(find==0)
                printf("没有找到信息! \n");
            break;

            case 2: printf("请输入姓名: ");
            scanf("%s",name);
            k2=hash2(name);
```

```
                    f=name_list[k2];
                    while (f!=NULL)
                    {
                        if(strcmp(f->name,name)==0)
                        {
                            printf("找到信息：号码-%s 姓名-%s 地址-%s\n",f->num,
f->name,f->address);
                            find=1;
                        }
                        f=f->next;
                    }
                    if(find==0)
                        printf("没有找到信息！\n");
                    break;
                }
            }

            int main()
            {
                int x,i;
                for(i=0;i<19;i++)
                {
                    num_list[i]=NULL;
                    name_list[i]=NULL;
                }
                while(1)
                {
                    printf("******************\n");
                    printf("1.创建电话号码表\n");
                    printf("2.查询\n");
                    printf("3.插入新号码\n");
                    printf("4.电话号码表显示\n");
                    printf("5.退出\n");
                    printf("请输入选项:");
                    scanf("%d",&x);
                    switch(x)
                    {
                        case 1: create();break;
                        case 2: search();break;
                        case 3: insert();break;
                        case 4: print();break;
                        case 5: return 0;
                        default:printf("请输入 1-5！\n");
```

```
            }
        }
    }
```



独立实践

如果采用开放定址法来解决冲突，则应该如何构造哈希表？

小 结

查找是数据处理中经常使用的一种技术，本章主要介绍了查找表的各种方法及查找效率的衡量标准——平均查找长度。查找过程中的主要操作是关键字和给定值进行的比较，因此以一次查找所需进行的比较次数的期望值作为查找方法效率的衡量标准，即平均查找长度。

线性表的查找主要介绍了顺序查找、二分查找和分块查找三种方法，如果线性表是有序的，那么二分查找是一种最快的查找方法。本章介绍了哈希表的概念、构造哈希函数和处理冲突的方法，哈希表方法是通过直接计算出节点的地址建立哈希表来进行查找的，不同于线性表查找的基于关键字的比较判断。对于现实的查找问题，应该根据问题的需要选择合适的查找方法及相应的存储结构。

习 题

一、填空题

1．在数据存放无规律的线性表中进行检索的最佳方法是________。

2．线性有序表(a_1，a_2，a_3，…，a_{256})是从小到大排列的，对一个给定的值 k，用二分法检索表中与 k 相等的元素，在查找不成功的情况下，最多需要检索________次。设有 100 个节点，用二分法查找时，最大比较次数是________。

3．假设在有序线性表 $a[20]$上进行二分查找，则比较一次查找成功的节点数为 1；比较两次查找成功的节点数为________；比较四次查找成功的节点数为________；平均查找长度为________。

4．二分查找有序表(4，6，12，20，28，38，50，70，88，100)，若查找表中元素 20，它将依次与表中元素________比较大小。

5．在各种查找方法中，平均查找长度与节点个数 n 无关的查找方法是________。

6．哈希法存储的基本思想是由________决定数据的存储地址。

7．有一个表长为 m 的散列表，初始状态为空，现将 $n(n<m)$个不同的关键码插入到哈希表中，解决冲突的方法是用线性探测法。如果这 n 个关键码的哈希地址都相同，则探测的总次数是________。

二、判断题

1．二分查找只能在有序的顺序表上进行而不能在有序链表上进行。 (　　)

2．二分查找的查找速度一定比顺序查找法的查找速度快。 (　　)

3．哈希表的查找效率完全取决于所选取的哈希函数和处理冲突的方法。 (　　)

4．有 n 个数存放在一维数组 $A[1\cdots n]$ 中，进行顺序查找时，这 n 个数的排列有序或无序其平均查找长度不同。 (　　)

5．适于对动态查找表进行高效率查找的组织结构是分块有序表。 (　　)

6．在哈希查找中，“比较”操作一般也是不可避免的。 (　　)

三、选择题

1．在表长为 n 的链表中进行线性查找，它的平均查找长度为________。

A．ASL=n　　B．ASL=$(n+1)/2$

C．ASL=$\sqrt{n}+1$　　D．ASL≈$\log_2(n+1)-1$

2．用二分查找法查找有序表(4，6，10，12，20，30，50，70，88，100)，若查找表中元素 58，则它将依次与表中(　　)比较大小，查找结果失败。

A．20，70，30，50　　B．30，88，70，50

C．20，50　　D．30，88，50

3．对 22 个记录的有序表做二分查找，当查找失败时，至少需要比较(　　)次关键字。

A．3　　B．4　　C．5　　D．6

4．链表适用于(　　)查找

A．顺序　　B．二分法

C．顺序，也能二分法　　D．随机

5．要进行线性查找，则线性表①(　　)；要进行二分查找，则线性表②(　　)；要进行哈希查找，则线性表③(　　)。

某顺序存储的表格，其中有 90000 个元素，已按关键项的值的上升顺序排列。现假定对各个元素进行查找的概率是相同的，并且各个元素的关键项的值皆不相同。当用顺序查找法查找时，平均比较次数约为④(　　)，最大比较次数为⑤(　　)。

供选择的答案：

①～③：A．必须以顺序方式存储　　B．必须以链表方式存储

C．必须以散列方式存储

D．既可以以顺序方式，也可以以链表方式存储

E．必须以顺序方式存储且数据元素已按值递增或递减的次序排好

F．必须以链表方式存储且数据元素已按值递增或递减的次序排好

④，⑤：　A．25000　　B．30000　　C．45000　　D．90000

答案：①=(　　)　②=(　　)　③=(　　)　④=(　　)　⑤=(　　)

6．与其他查找方法相比，哈希查找法的特点是(　　)。

A．通过关键字的比较进行查找

B．通过关键字计算元素的存储地址进行查找

C．通过关键字计算元素的存储地址并进行一定的比较进行查找

D．以上都不是

7．静态查找表与动态查找表的根本区别在于(　　)。

A．它们的逻辑结构不一样　　B．施加在其上的操作不一样

C．所包含的数据元素类型不一样　　D．存储实现不一样

四、简答题

1．二分查找适不适合链表结构的序列，为什么？用二分查找法的查找速度必然比线性查找的速度快，这种说法对吗？

2．假定对有序表(3，4，5，7，24，30，42，54，63，72，87，95)进行二分查找，试回答下列问题：

(1) 画出描述二分查找过程的判定树。

(2) 若查找元素54，需依次与哪些元素比较？

(3) 若查找元素90，需依次与哪些元素比较？

(4) 假定每个元素的查找概率相等，求查找成功时的平均查找长度。

3．设哈希表的地址范围为0～17，哈希函数为 $H(K)=K \mod 16$。

K 为关键字，用线性探测法再散列法处理冲突，输入关键字序列：

(10，24，32，17，31，30，46，47，40，63，49)

构造哈希表，试回答下列问题：

(1) 画出哈希表的示意图。

(2) 若查找关键字63，需要依次与哪些关键字进行比较？

(3) 若查找关键字60，需要依次与哪些关键字比较？

(4) 假定每个关键字的查找概率相等，求查找成功时的平均查找长度。

4．编写一个算法，利用二分查找算法在一个有序表中插入一个元素 x，并保持表的有序性。

5．选取哈希函数 $H(key)=(3*key)\%11$，用线性探测法处理冲突，对下列关键码序列构造一个散列地址空间为0～10，表长为11的哈希表，{22，41，53，08，46，30，01，31，66}。

参 考 文 献

[1] 严蔚敏，吴伟民. 数据结构(C 语言版). 北京：清华大学出版社，1997.

[2] 唐国民，王国均. 数据结构(C 语言版). 北京：清华大学出版社，2009.

[3] [美]Mark Allen Weiss. 数据结构与算法分析—C 语言描述(原书第 2 版). 冯舜玺，译. 北京：机械工业出版社，2004.

[4] [美]Brain W.Kernighan, Dennis M.Ritchie. C 程序设计语言. 2 版. 徐宝文，等译. 北京：机械工业出版社，2004.

[5] 陈守孔，等. 算法与数据结构考研试题精析. 2 版. 北京：机械工业出版社，2007.

[6] 苏仕华. 数据结构课程设计. 北京：机械工业出版社，2010.

[7] 马巧梅. 数据结构课程设计案例教程.北京：人民邮电出版社，2012.

[8] 程杰. 大话数据结构. 北京：清华大学出版社，2011.

[9] 陈明. 数据结构(C 语言描述). 2 版.北京：清华大学出版社，2011.

[10] 王国均、唐国民.数据结构实验教程(C 语言). 北京：清华大学出版社，2009.

北京大学出版社本科计算机系列实用规划教材

序号	标准书号	书　　名	主　编	定价	序号	标准书号	书　　名	主　编	定价
1	7-301-10511-5	离散数学	段禅伦	28	38	7-301-13684-3	单片机原理及应用	王新颖	25
2	7-301-10457-X	线性代数	陈付贵	20	39	7-301-14505-0	Visual C++程序设计案例教程	张荣梅	30
3	7-301-10510-X	概率论与数理统计	陈荣江	26	40	7-301-14259-2	多媒体技术应用案例教程	李　建	30
4	7-301-10503-0	Visual Basic 程序设计	闵联营	22	41	7-301-14503-6	ASP .NET 动态网页设计案例教程(Visual Basic .NET 版)	江　红	35
5	7-301-21752-8	多媒体技术及其应用(第 2 版)	张　明	39	42	7-301-14504-3	C++面向对象与 Visual C++程序设计案例教程	黄贤英	35
6	7-301-10466-8	C++程序设计	刘天印	33	43	7-301-14506-7	Photoshop CS3 案例教程	李建芳	34
7	7-301-10467-5	C++程序设计实验指导与习题解答	李　兰	20	44	7-301-14510-4	C++程序设计基础案例教程	于永彦	33
8	7-301-10505-4	Visual C++程序设计教程与上机指导	高志伟	25	45	7-301-14942-3	ASP .NET 网络应用案例教程(C# .NET 版)	张登辉	33
9	7-301-10462-0	XML 实用教程	丁跃潮	26	46	7-301-12377-5	计算机硬件技术基础	石　磊	26
10	7-301-10463-7	计算机网络系统集成	斯桃枝	22	47	7-301-15208-9	计算机组成原理	娄国焕	24
11	7-301-22437-3	单片机原理及应用教程(第 2 版)	范立南	43	48	7-301-15463-2	网页设计与制作案例教程	房爱莲	36
12	7-5038-4421-3	ASP .NET 网络编程实用教程(C#版)	崔良海	31	49	7-301-04852-8	线性代数	姚喜妍	22
13	7-5038-4427-2	C 语言程序设计	赵建锋	25	50	7-301-15461-8	计算机网络技术	陈代武	33
14	7-5038-4420-5	Delphi 程序设计基础教程	张世明	37	51	7-301-15697-1	计算机辅助设计二次开发案例教程	谢安俊	26
15	7-5038-4417-5	SQL Server 数据库设计与管理	姜　力	31	52	7-301-15740-4	Visual C# 程序开发案例教程	韩朝阳	30
16	7-5038-4424-9	大学计算机基础	贾丽娟	34	53	7-301-16597-3	Visual C++程序设计实用案例教程	于永彦	32
17	7-5038-4430-0	计算机科学与技术导论	王昆仑	30	54	7-301-16850-9	Java 程序设计案例教程	胡巧多	32
18	7-5038-4418-3	计算机网络应用实例教程	魏　峥	25	55	7-301-16842-4	数据库原理与应用(SQL Server 版)	毛一梅	36
19	7-5038-4415-9	面向对象程序设计	冷英男	28	56	7-301-16910-0	计算机网络技术基础与应用	马秀峰	33
20	7-5038-4429-4	软件工程	赵春刚	22	57	7-301-15063-4	计算机网络基础与应用	刘远生	32
21	7-5038-4431-0	数据结构(C++版)	秦　锋	28	58	7-301-15250-8	汇编语言程序设计	张光长	28
22	7-5038-4423-2	微机应用基础	吕晓燕	33	59	7-301-15064-1	网络安全技术	骆耀祖	30
23	7-5038-4426-4	微型计算机原理与接口技术	刘彦文	26	60	7-301-15584-4	数据结构与算法	佟伟光	32
24	7-5038-4425-6	办公自动化教程	钱　俊	30	61	7-301-17087-8	操作系统实用教程	范立南	36
25	7-5038-4419-1	Java 语言程序设计实用教程	董迎红	33	62	7-301-16631-4	Visual Basic 2008 程序设计教程	隋晓红	34
26	7-5038-4428-0	计算机图形技术	龚声蓉	28	63	7-301-17537-8	C 语言基础案例教程	汪新民	31
27	7-301-11501-5	计算机软件技术基础	高　巍	25	64	7-301-17397-8	C++程序设计基础教程	郗亚辉	30
28	7-301-11500-8	计算机组装与维护实用教程	崔明远	33	65	7-301-17578-1	图论算法理论、实现及应用	王桂平	54
29	7-301-12174-0	Visual FoxPro 实用教程	马秀峰	29	66	7-301-17964-2	PHP 动态网页设计与制作案例教程	房爱莲	42
30	7-301-11500-8	管理信息系统实用教程	杨月江	27	67	7-301-18514-8	多媒体开发与编程	于永彦	35
31	7-301-11445-2	Photoshop CS 实用教程	张　瑾	28	68	7-301-18538-4	实用计算方法	徐亚平	24
32	7-301-12378-2	ASP .NET 课程设计指导	潘志红	35	69	7-301-18539-1	Visual FoxPro 数据库设计案例教程	谭红杨	35
33	7-301-12394-2	C# .NET 课程设计指导	龚自霞	32	70	7-301-19313-6	Java 程序设计案例教程与实训	董迎红	45
34	7-301-13259-3	VisualBasic .NET 课程设计指导	潘志红	30	71	7-301-19389-1	Visual FoxPro 实用教程与上机指导（第 2 版）	马秀峰	40
35	7-301-12371-3	网络工程实用教程	汪新民	34	72	7-301-19435-5	计算方法	尹景本	28
36	7-301-14132-8	J2EE 课程设计指导	王立丰	32	73	7-301-19388-4	Java 程序设计教程	张剑飞	35
37	7-301-21088-8	计算机专业英语(第 2 版)	张　勇	42	74	7-301-19386-0	计算机图形技术(第 2 版)	许承东	44

序号	标准书号	书　　名	主　编	定价	序号	标准书号	书　　名	主　编	定价
75	7-301-15689-6	Photoshop CS5 案例教程(第 2 版)	李建芳	39	84	7-301-16824-0	软件测试案例教程	丁宋涛	28
76	7-301-18395-3	概率论与数理统计	姚喜妍	29	85	7-301-20328-6	ASP. NET 动态网页案例教程(C#.NET 版)	江　红	45
77	7-301-19980-0	3ds Max 2011 案例教程	李建芳	44	86	7-301-16528-7	C#程序设计	胡艳菊	40
78	7-301-20052-0	数据结构与算法应用实践教程	李文书	36	87	7-301-21271-4	C#面向对象程序设计及实践教程	唐　燕	45
79	7-301-12375-1	汇编语言程序设计	张宝剑	36	88	7-301-21295-0	计算机专业英语	吴丽君	34
80	7-301-20523-5	Visual C++程序设计教程与上机指导(第 2 版)	牛江川	40	89	7-301-21341-4	计算机组成与结构教程	姚玉霞	42
81	7-301-20630-0	C#程序开发案例教程	李挥剑	39	90	7-301-21367-4	计算机组成与结构实验实训教程	姚玉霞	22
82	7-301-20898-4	SQL Server 2008 数据库应用案例教程	钱哨	38	91	7-301-22119-8	UML 实用基础教程	赵春刚	36
83	7-301-21052-9	ASP.NET 程序设计与开发	张绍兵	39	92	7-301-22965-1	数据结构(C 语言版)	陈超祥	32

北京大学出版社电气信息类教材书目(已出版)

欢迎选订

序号	标准书号	书　名	主　编	定价	序号	标准书号	书　名	主　编	定价
1	7-301-10759-1	DSP 技术及应用	吴冬梅	26	38	7-5038-4400-3	工厂供配电	王玉华	34
2	7-301-10760-7	单片机原理与应用技术	魏立峰	25	39	7-5038-4410-2	控制系统仿真	郑恩让	26
3	7-301-10765-2	电工学	蒋　中	29	40	7-5038-4398-3	数字电子技术	李　元	27
4	7-301-19183-5	电工与电子技术(上册)(第2版)	吴舒辞	30	41	7-5038-4412-6	现代控制理论	刘永信	22
5	7-301-19229-0	电工与电子技术(下册)(第2版)	徐卓农	32	42	7-5038-4401-0	自动化仪表	齐志才	27
6	7-301-10699-0	电子工艺实习	周春阳	19	43	7-5038-4408-9	自动化专业英语	李国厚	32
7	7-301-10744-7	电子工艺学教程	张立毅	32	44	7-5038-4406-5	集散控制系统	刘翠玲	25
8	7-301-10915-6	电子线路 CAD	吕建平	34	45	7-301-19174-3	传感器基础(第 2 版)	赵玉刚	32
9	7-301-10764-1	数据通信技术教程	吴延海	29	46	7-5038-4396-9	自动控制原理	潘　丰	32
10	7-301-18784-5	数字信号处理(第 2 版)	阎　毅	32	47	7-301-10512-2	现代控制理论基础(国家级十一五规划教材)	侯媛彬	20
11	7-301-18889-7	现代交换技术(第 2 版)	姚　军	36	48	7-301-11151-2	电路基础学习指导与典型题解	公茂法	32
12	7-301-10761-4	信号与系统	华　容	33	49	7-301-12326-3	过程控制与自动化仪表	张井岗	36
13	7-301-19318-1	信息与通信工程专业英语（第 2 版）	韩定定	32	50	7-301-12327-0	计算机控制系统	徐文尚	28
14	7-301-10757-7	自动控制原理	袁德成	29	51	7-5038-4414-0	微机原理及接口技术	赵志诚	38
15	7-301-16520-1	高频电子线路(第 2 版)	宋树祥	35	52	7-301-10465-1	单片机原理及应用教程	范立南	30
16	7-301-11507-7	微机原理与接口技术	陈光军	34	53	7-5038-4426-4	微型计算机原理与接口技术	刘彦文	26
17	7-301-11442-1	MATLAB 基础及其应用教程	周开利	24	54	7-301-12562-5	嵌入式基础实践教程	杨　刚	30
18	7-301-11508-4	计算机网络	郭银景	31	55	7-301-12530-4	嵌入式 ARM 系统原理与实例开发	杨宗德	25
19	7-301-12178-8	通信原理	隋晓红	32	56	7-301-13676-8	单片机原理与应用及 C51 程序设计	唐　颖	30
20	7-301-12175-7	电子系统综合设计	郭　勇	25	57	7-301-13577-8	电力电子技术及应用	张润和	38
21	7-301-11503-9	EDA 技术基础	赵明富	22	58	7-301-20508-2	电磁场与电磁波（第 2 版）	邬春明	30
22	7-301-12176-4	数字图像处理	曹茂永	23	59	7-301-12179-5	电路分析	王艳红	38
23	7-301-12177-1	现代通信系统	李白萍	27	60	7-301-12380-5	电子测量与传感技术	杨　雷	35
24	7-301-12340-9	模拟电子技术	陆秀令	28	61	7-301-14461-9	高电压技术	马永翔	28
25	7-301-13121-3	模拟电子技术实验教程	谭海曙	24	62	7-301-14472-5	生物医学数据分析及其 MATLAB 实现	尚志刚	25
26	7-301-11502-2	移动通信	郭俊强	22	63	7-301-14460-2	电力系统分析	曹　娜	35
27	7-301-11504-6	数字电子技术	梅开乡	30	64	7-301-14459-6	DSP 技术与应用基础	俞一彪	34
28	7-301-18860-6	运筹学(第 2 版)	吴亚丽	28	65	7-301-14994-2	综合布线系统基础教程	吴达金	24
29	7-5038-4407-2	传感器与检测技术	祝诗平	30	66	7-301-15168-6	信号处理 MATLAB 实验教程	李　杰	20
30	7-5038-4413-3	单片机原理及应用	刘　刚	24	67	7-301-15440-3	电工电子实验教程	魏　伟	26
31	7-5038-4409-6	电机与拖动	杨天明	27	68	7-301-15445-8	检测与控制实验教程	魏　伟	24
32	7-5038-4411-9	电力电子技术	樊立萍	25	69	7-301-04595-4	电路与模拟电子技术	张绪光	35
33	7-5038-4399-0	电力市场原理与实践	邹　斌	24	70	7-301-15458-8	信号、系统与控制理论(上、下册)	邱德润	70
34	7-5038-4405-8	电力系统继电保护	马永翔	27	71	7-301-15786-2	通信网的信令系统	张云麟	24
35	7-5038-4397-6	电力系统自动化	孟祥忠	25	72	7-301-16493-8	发电厂变电所电气部分	马永翔	35
36	7-5038-4404-1	电气控制技术	韩顺杰	22	73	7-301-16076-3	数字信号处理	王震宇	32
37	7-5038-4403-4	电器与 PLC 控制技术	陈志新	38	74	7-301-16931-5	微机原理及接口技术	肖洪兵	32

序号	标准书号	书　　名	主　编	定价	序号	标准书号	书　　名	主　编	定价
75	7-301-16932-2	数字电子技术	刘金华	30	110	7-301-20845-8	单片机原理与接口技术实验与课程设计	徐懂理	26
76	7-301-16933-9	自动控制原理	丁　红	32	111	301-20725-3	模拟电子线路	宋树祥	38
77	7-301-17540-8	单片机原理及应用教程	周广兴	40	112	7-301-21058-1	单片机原理与应用及其实验指导书	邵发森	44
78	7-301-17614-6	微机原理及接口技术实验指导书	李干林	22	113	7-301-20918-9	Mathcad 在信号与系统中的应用	郭仁春	30
79	7-301-12379-9	光纤通信	卢志茂	28	114	7-301-20327-9	电工学实验教程	王士军	34
80	7-301-17382-4	离散信息论基础	范九伦	25	115	7-301-16367-2	供配电技术	王玉华	49
81	7-301-17677-1	新能源与分布式发电技术	朱永强	32	116	7-301-20351-4	电路与模拟电子技术实验指导书	唐　颖	26
82	7-301-17683-2	光纤通信	李丽君	26	117	7-301-21247-9	MATLAB 基础与应用教程	王月明	32
83	7-301-17700-6	模拟电子技术	张绪光	36	118	7-301-21235-6	集成电路版图设计	陆学斌	36
84	7-301-17318-3	ARM 嵌入式系统基础与开发教程	丁文龙	36	119	7-301-21304-9	数字电子技术	秦长海	49
85	7-301-17797-6	PLC 原理及应用	缪志农	26	120	7-301-21366-7	电力系统继电保护(第 2 版)	马永翔	42
86	7-301-17986-4	数字信号处理	王玉德	32	121	7-301-21450-3	模拟电子与数字逻辑	邬春明	39
87	7-301-18131-7	集散控制系统	周荣富	36	122	7-301-21439-8	物联网概论	王金甫	42
88	7-301-18285-7	电子线路 CAD	周荣富	41	123	7-301-21849-5	微波技术基础及其应用	李泽民	49
89	7-301-16739-7	MATLAB 基础及应用	李国朝	39	124	7-301-21688-0	电子信息与通信工程专业英语	孙桂芝	36
90	7-301-18352-6	信息论与编码	隋晓红	24	125	7-301-22110-5	传感器技术及应用电路项目化教程	钱裕禄	30
91	7-301-18260-4	控制电机与特种电机及其控制系统	孙冠群	42	126	7-301-21672-9	单片机系统设计与实例开发（MSP430）	顾　涛	44
92	7-301-18493-6	电工技术	张　莉	26	127	7-301-22112-9	自动控制原理	许丽佳	30
93	7-301-18496-7	现代电子系统设计教程	宋晓梅	36	128	7-301-22109-9	DSP 技术及应用	董　胜	39
94	7-301-18672-5	太阳能电池原理与应用	靳瑞敏	25	129	7-301-21607-1	数字图像处理算法及应用	李文书	48
95	7-301-18314-4	通信电子线路及仿真设计	王鲜芳	29	130	7-301-22111-2	平板显示技术基础	王丽娟	52
96	7-301-19175-0	单片机原理与接口技术	李　升	46	131	7-301-22448-9	自动控制原理	谭功全	44
97	7-301-19320-4	移动通信	刘维超	39	132	7-301-22474-8	电子电路基础实验与课程设计	武　林	36
98	7-301-19447-8	电气信息类专业英语	缪志农	40	133	7-301-22484-7	电文化——电气信息学科概论	高　心	30
99	7-301-19451-5	嵌入式系统设计及应用	邢吉生	44	134	7-301-22436-6	物联网技术案例教程	崔逊学	40
100	7-301-19452-2	电子信息类专业 MATLAB 实验教程	李明明	42	135	7-301-22598-1	实用数字电子技术	钱裕禄	30
101	7-301-16914-8	物理光学理论与应用	宋贵才	32	136	7-301-22529-5	PLC 技术与应用(西门子版)	丁金婷	32
102	7-301-16598-0	综合布线系统管理教程	吴达金	39	137	7-301-22386-4	自动控制原理	佟　威	30
103	7-301-20394-1	物联网基础与应用	李蔚田	44	138	7-301-22528-8	通信原理实验与课程设计	邬春明	34
104	7-301-20339-2	数字图像处理	李云红	36	139	7-301-22582-0	信号与系统	许丽佳	38
105	7-301-20340-8	信号与系统	李云红	29	140	7-301-22447-2	嵌入式系统基础实践教程	韩　磊	35
106	7-301-20505-1	电路分析基础	吴舒辞	38	141	7-301-22776-3	信号与线性系统	朱明旱	33
107	7-301-22447-2	嵌入式系统基础实践教程	韩　磊	35	142	7-301-22872-2	电机、拖动与控制	万芳瑛	34
108	7-301-20506-8	编码调制技术	黄　平	26	143	7-301-22882-1	MCS-51 单片机原理及应用	黄翠翠	34
109	7-301-20763-5	网络工程与管理	谢　慧	39	144	7-301-22936-1	自动控制原理	邢春芳	39

相关教学资源如电子课件、电子教材、习题答案等可以登录 www.pup6.com 下载或在线阅读。

扑六知识网(www.pup6.com)有海量的相关教学资源和电子教材供阅读及下载(包括北京大学出版社第六事业部的相关资源)，同时欢迎您将教学课件、视频、教案、素材、习题、试卷、辅导材料、课改成果、设计作品、论文等教学资源上传到 pup6.com，与全国高校师生分享您的教学成就与经验，并可自由设定价格，知识也能创造财富。具体情况请登录网站查询。

如您需要免费纸质样书用于教学，欢迎登陆第六事业部门户网(www.pup6.com)填表申请，并欢迎在线登记选题以到北京大学出版社来出版您的大作，也可下载相关表格填写后发到我们的邮箱，我们将及时与您取得联系并做好全方位的服务。

扑六知识网将打造成全国最大的教育资源共享平台，欢迎您的加入——让知识有价值，让教学无界限，让学习更轻松。

联系方式：010-62750667，pup6_czq@163.com，szheng_pup6@163.com，linzhangbo@126.com，欢迎来电来信咨询。